Porting Rodinia Applications to OmpSs@FPGA

Marc Perelló Bacardit

A thesis presented for the bachelor degree of Informatics Engineering

Date: October 22nd, 2019
Director: Xavier Martorell Bofill
Department: Arquitectura de Computadors
Specialty: Computer Engineering

FACULTAT D'INFORMÀTICA DE BARCELONA(FIB)
UNIVERSITAT POLITÈCNICA DE BARCELONA(UPC) - BarcelonaTech

Contents

List of Tables

List of Figures

List of Codes

Resum

La computació heterogènia amb FPGAs és una alternativa de baix consum a altres sistemes usats freqüentment, com la computació amb CPU multi-nucli i la computació heterogènia amb GPUs. No obstant, degut a que les FPGAs funcionen d'una manera totalment diferent a altres dispositius fets servir en computació, són bastant difícils de comparar.

La Rodinia Benchmark Suite està formada per aplicacions que poden usar-se per comparar sistemes de computació heterogenis. La suite ha adaptat les aplicacions per computació amb CPU multi-nucli i computació amb GPU (fent servir les llibreries OpenMP, Cuda, OpenCL).

L'objectiu del projecte és adaptar un cert nombre d'aquestes aplicacions per OmpSs@FPGA, un sistema de computació heterogeni amb dispositius FPGA de Xilinx. Algunes d'aquestes aplicacions també seran optimitzades fent servir eines de OmpSs i de Xilinx (Vivado HLS).

Tot i que al principi la idea era adaptar i provar les aplicacions en el dispositiu FPGA físic, la absència del hardware durant la primera part de la fase d'adaptació va incentivar el desenvolupament d'un entorn de simulació de dispositius FPGA. Tal cosa va implicar modificar el runtime per fer que es comuniqués amb un programa software enlloc d'intentar accedir al hardware real. Aquesta tasca va afegir una càrrega de treball considerable en el projecte que no estava prevista. Tot i així, degut a que aquest entorn de simulació va fer molt més ràpida l'adaptació de les aplicacions, la quantitat d'hores amb les que es va desenvolupar l'entorn i es van adaptar les aplicacions va coincidir amb les hores previstes inicialment només per l'adaptació.

Es van adaptar un total de 7 aplicacions, 6 de les quals es van optimitzar fins a cert punt. També es van analitzar totes les optimitzacions parcials acumulatives fent servir traces d'execució visualitzades amb el software Paraver. Un cop fets els anàlisis, es va fer un informe de sostenibilitat per avaluar l'impacte del projecte en els aspectes ambiental, econòmic i social.

Finalment, s'arriba a la conclusió que s'ha completat l'objectiu inicial del projecte satisfactòriament.

Resumen

La computación heterogénea con FPGAs es una alternativa de bajo consumo a otros sistemas usados normalmente, como la computación con CPU multi-núcleo y la computación heterogénea con GPUs. No obstante, debido a que las FPGAs funcionan de una manera totalmente diferente a los otros dispositivos usados en computación, son bastante difíciles de comparar.

La Rodinia Benchmark Suite está compuesta por aplicaciones que pueden usarse para comparar sistemas de computación heterogéneos. La suite ha adaptado dichas aplicaciones para computación con CPU multi-núcleo y computación heterogénea con GPU (usando las librerías OpenMP, Cuda y OpenCL).

El objetivo del proyecto es adaptar un subconjunto de estas aplicaciones para OmpSs@FPGA, un sistema de computación heterogénea con dispositivos FPGA de Xilinx. Algunas de estas aplicaciones también se optimizarán usando herramientas de OmpSs y de Xilinx (Vivado HLS).

Aunque al principio la idea era adaptar y probar las aplicaciones en un dispositivo FPGA físico, la ausencia del hardware durante la primera parte de la fase de adaptación incentivó el desarrollo de un entorno de simulación de dispositivos FPGA. Eso implicó modificar el runtime para hacer que se comunicase con un programa software en vez de intentar acceder al hardware real. Esta tarea añadió una carga de trabajo considerable en el proyecto que no estaba prevista. Aun así, debido a que este entorno de simulación hizo mucho más rápida la adaptación de las aplicaciones, el número de horas con las que se desarrolló el entorno y se adaptaron las aplicaciones coincidió con las horas previstas inicialmente para solo la adaptación.

Se adaptaron un total de 7 aplicaciones, 6 de las cuales se optimizaron hasta cierto punto. También se analizaron todas las optimizaciones parciales acumulativas usando trazas de ejecución visualizadas con el software Paraver. Una vez hechos los análisis, se hizo un informe de sostenibilidad para evaluar el impacto del proyecto en los aspectos ambiental, económico y social.

Finalmente, se concluye que el objetivo inicial del proyecto se ha alcanzado, y por tanto el proyecto se ha completado satisfactoriamente.

Abstract

FPGA computing is a low power alternative to the vastly used multi-core CPU and GPU computing systems. However, due to FPGA devices being completely different in terms of architecture, they are quite complex to compare to other forms of computing.

Rodinia Benchmark Suite consists of a number of applications that can be used to benchmark heterogeneous computing systems. The suite has currently adapted the applications for multi-core CPU and GPU computing (using OpenMP, Cuda and OpenCL libraries).

The objective of this project is to port some of the applications from the Rodinia Benchmark Suite to OmpSs@FPGA, a heterogeneous FPGA computing environment based on Xilinx FPGA devices. A portion of these applications will also be optimized using both OmpSs features and Xilinx tools (Vivado HLS).

While the original intentions were to port and test the applications with a physical FPGA device, the lack of access to the hardware during the initial porting phase encouraged the development of a simulated FPGA environment. This implied modifying the runtime to communicate with a software block running as an executable instead of trying to access the real hardware. Even though it added a significant workload to the project that was not intended at first, it ended up making the porting of the applications much faster than with the real hardware. Ultimately, the expected number of hours from the initial planning matched the hours it took to both develop the simulated environment and the applications.

A total of 7 applications were ported to the OmpSs@FPGA environment, 6 of which were optimized to a certain extent. Furthermore, each of the accumulated optimization stages for every optimized application was analyzed and explained using Paraver traces. After that, a sustainability report was made in order to evaluate the impact of the project environmentally, economically and socially wise.

In the final conclusions, it is stated that the original objective of the project has been fulfilled and thus the project has been completed successfully.

Chapter 1

Context and Scope of the project

1.1 Introduction

Hardware acceleration has been a fundamental part in a lot of technological and scientific fields. Since computer CPUs (Central Processing Unit) are general-purpose, it is common practice to speed up certain sections of an algorithm using specialized computing devices called accelerators. This is called heterogeneous computing, since it involves more than one type of processing unit in order to solve a problem. Most widely known examples of accelerators are GPUs (Graphics Processing Unit), co-processors and FPGAs (Field Programmable Gate Array).

CPUs and GPUs are instruction set architectures. Instruction set architectures are composed of a limited set of instructions which they understand and execute in order to provide the necessary functionalities.

FPGAs, unlike CPUs and GPUs, are not instruction set architectures. They consist of a matrix of configurable logic blocks (CLBs) which can be configured depending on what your application requires [1]. That means they can be programmed to simulate specific integrated circuits in order to compute a specific computation with less latency than an instruction based accelerator would. They are also more flexible, since you can decide which circuit to simulate, thus being able to decide which latency, power consumption [2] and area (amount of CLBs they occupy) you want your design to have depending on your specific needs for your application [3].

FPGAs, however, have a significant disadvantage: they are much more complex to program than any instruction set architecture processor. This leads

to FPGA programming being difficult to optimize without one or several layers of abstraction in between.

In order to program the FPGA, you need a sequence of bits called bitstream. The bitstream contains all the information about the configurable logic blocks and other data inside the FPGA for the specific integrated circuit you want to simulate. Since this part is very low level and device dependent, it's usually generated by proprietary applications from FPGA manufacturers using a HDL(Hardware Description Language) or even a high level language if supported.

This project is going to be focused on FPGA programming. More precisely, it's going to be focused on Xilinx based FPGAs. Using an extension of OpenMP called OmpSs, the objective of the project is to port a set of applications from the Rodinia Benchmark Suite to the OmpSs@FPGA heterogeneous computing environment.

1.2 Stakeholders

1.2.1 Project director

As it's usually the case with final grade projects, the idea of the project itself was given by the project director, and gets benefits from it, along with the Departament d'Arquitectura de Computadors of the UPC, and the Barcelona Supercomputing Center, as hosts of his research.

1.2.2 Wide range of studies

The target range for this project is as diverse as the Rodina Benchmark suite, since it adds an additional accelerator type in order to give a more precise insight about the best performance on each domain. There are applications for medical imaging, bioinformatics, linear algebra, physics simulation, graph algorithms, image processing and molecular dynamics.

1.3 State of the art

1.3.1 OmpSs@FPGA

OmpSs is an extension of the OpenMP library which adds new directives to support asynchronous parallelism and heterogeneity [4]. This enables the possibility to use accelerators like GPUs or FPGAs in parallel executions while keeping the code

as simple as possible. It's fully task-based, meaning parallel directives are not supported. OmpSs is compatible with C/C++ languages and is compiled with their own compiler named Mercurium. OmpSs@FPGA is an extension of OmpSs that is focused on supporting FPGA acceleration on Xilinx devices [5].

1.3.2 Rodinia Benchmark Suite

With the emerge of multi-core architectures, a lot of benchmarks have been created for comparing and analyzing parallel performance. However, most of them only compare general purpose CPU architectures. Rodinia Benchmark Suite sees the need for new heterogeneous computing benchmarking due to the rise of heterogenic systems in order to improve tasks efficiency. In the paper, they analyze a number of applications using multi-core CPUs and GPUs, using OpenMP and Cuda libraries respectively. They choose numerous applications with a wide range of different domains of study, in order to give insight of the difference between those 2 architectures [6].

Outside of the paper, throughout the years, they extended the amount of applications and also supported OpenCL library for most of them. However, there still hasn't been any implementations of FPGA acceleration.

1.4 Scope

The Rodinia Benchmark Suite is used to learn about architectural differences between CPUs and GPUs and their performance in several domains of study (bioinformatics, physics, medical imaging, etc.).

OmpSs environment is composed of several heterogeneous systems. The idea is to extend Rodinia Benchmark Suite to FPGA accelerators so that it can be added to OmpSs@FPGA heterogeneous system, in order to further analyze the performance of FPGA based systems in all those domains of study in comparison to CPUs and GPUs.

After the porting of the applications is done, a certain number of them will be analyzed and further optimized if possible.

Given how different FPGA systems work in comparison to CPU and GPU systems, the addition of those in the benchmark could give really interesting insight of their performance compared to CPU/GPUs in different fields of study.

1.5 Methodology

As it is common with software based projects, the progress and evolution of our initial idea will depend on many factors and be subjected to as many changes as deemed necessary. In other words, our decisions and priorities about the porting and optimization of applications might be altered from the initial intentions if we believe it helps improve the overall project.

Most current methodologies have a rather fixed plan development, and little to no room for any changes that need to be made on the fly. The agile approach, on the other hand, has the flexibility and fast paced development which is ideal for our project.

While we cannot say our methodology matches exactly any of the methodologies based on the agile approach, since they are more team-oriented, we still follow some important aspects that make our project flexible and fast paced.

1.5.1 Regular planning and tuning

Because of the nature of most software projects, the next steps will depend significantly on the results of the previous phase. If the result is not the expected or could use some improvements, the project will be adjusted accordingly. For this to be possible, there needs to be discussions and planning meetings on a regular basis.

1.5.2 Client feedback dependent

The main stakeholder (project director) will take part in the meetings and will be of great importance for the project development. His decisions and opinions will heavily influence the planning of the project.

1.5.3 Frequent software development

Since there will be potential change of plans on a regular basis, the development of the project needs to keep up with that pace. Development needs to be done as soon as possible so that it can be fine-tuned for a better final outcome.

1.5.4 Based on working results

The project will be heavily reliable on porting results. Without a good approach on the software development, the final results might not determine anything or even arrive to the wrong conclusions. This is one of the reasons why there must be a reliable validation method for every ported application.

1.6 Possible obstacles and solutions

1.6.1 Bad planning

Bad planning can easily lead to problematic results, even if we plan every next phase on the fly. Because of this, the planning for each phase needs to be well thought through and as detailed as possible before we start executing it.

1.6.2 Bugs in the porting

As with any software related projects, we can expect the possibility of a software error to appear while porting one or several applications from the Rodinia Benchmark Suite to the FPGA environment. This is the reason why there needs to be a complete test bench for each ported application, which must include both average and extreme cases.

For the porting to be considered bug and error free, both the original and the ported application must give the same result for every input in the test bench.

1.6.3 Capacity issues in the FPGA

Since the number of CLBs in a FPGA is limited, it's feasible to consider the possibility of a certain hardware design to be bigger than the actual FPGA capabilities. While the possibility of getting a bigger FPGA exists, it's not cost efficient to do so in the middle of the project.

Instead, thanks to Vivado HLS tools, we are going to purposely shrink the hardware design enough so that it fits in our FPGA hardware. This process, however, could decrease the performance of the simulated circuit by a fair margin, so we will need to take this in consideration throughout the project.

1.7 Development tools

1.7.1 Vivado HLS (High Level Synthesis)

Works on a high level of abstraction, and enables the use of C/C++/SystemC languages to program their Xilinx devices instead of having to use HDL languages. Thanks to this feature, OmpSs@FPGA can easily integrate the original code into a Vivado HLS project, and can also be further optimized by HLS directives by the developer if needed [3].

1.7.2 Vivado

Works on a low level of abstraction and is used for designing the hardware you want the application to have, plus defining its behavior using an HDL language. It's also responsible for generating the bitstream. In our case, this process will be automatically done by the OmpSs@FPGA tools.

1.7.3 AutoVivado

AutoVivado is an essential tool for OmpSs@FPGA environment since it's used to create the Vivado and Vivado HLS projects from the original code. It's also used for automatically generating the bitstream from the Vivado project.

1.7.4 Mercurium

As mentioned earlier, Mercurium is the OmpSs compiler. If Mercurium detects FPGA targeting in the code, it separates that section from the rest so that AutoVivado can generate all the necessary files for Vivado and Vivado HLS projects in order to generate the bitstream for the FPGA.

On the other hand, the CPU based code is compiled and then executed on the host. It is then executed and managed by the OmpSs runtime called Nanos++ [7].

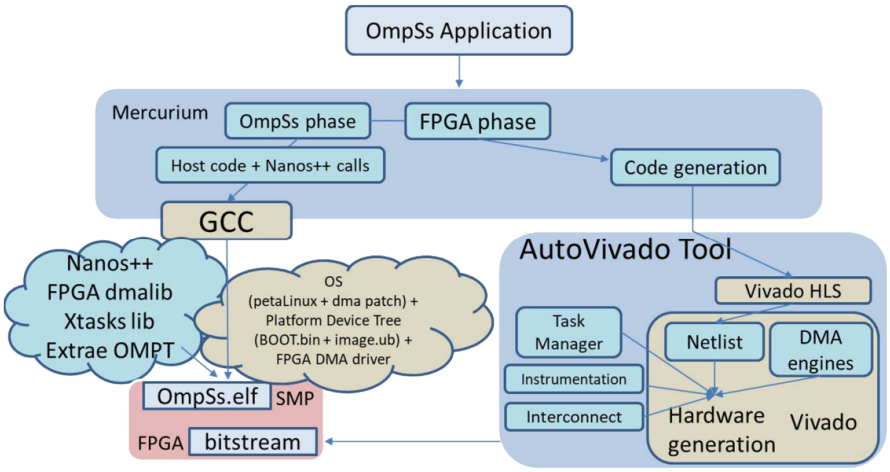


Figure 1.1: OmpSs@FPGA compilation diagram

1.7.5 Nanos++

Nanos++ is a runtime designed to support parallel environments. It's mainly used to support OmpSs, however it can also support OpenMP. Nanos++ provides support for task parallelism using synchronizations based on data dependences. It also comes with instrumentation that allows to obtain traces which can be used to analyze the performance of parallel applications.

1.7.6 Petalinux

Petalinux is a set of tools that makes easier the creation and configuration of the boot system and Linux environment for Xilinx based boards. You can customize your boot and Linux operative system to fit your needs and also has the option of emulating the boot and execution of your board using one of available emulators in the Petalinux installation. In our case, we'll use the emulator QEMU to test our first steps of the porting using a Petalinux boot along with our Linux environment.

1.8 Validation method

1.8.1 Plan validation

In order to keep the project in the right direction, there will be constant meetings with the project director as we mentioned earlier. Meetings will be mostly biweekly and will discuss how the project is going, the results of the last phase and what the next phase should be.

1.8.2 Software validation

Since we need to ensure the correct behavior of our porting on every application, we'll need to create a proper and extensive test bench that confirms both the original application and the ported application give the same results.

Chapter 2

Project planning

2.1 General planning

2.1.1 Estimated project duration

The project starts on February 19th of 2019 and will end on June 4th of 2019. It contains approximately 530 hours of work, which will be distributed in 40 hours per week on average. Each task has a weight on the total work amount proportional to the effort required.

2.1.2 Action plan

The project plan and duration assumes everything goes right and there are no major problems or time delays. If an obstacle appears during any of the tasks, however, there needs to be a fast re-routing or problem solving so that the project can resume its expected pace.

If that happens, extraordinary meetings will be held in order to handle the issue as fast as possible. If the obstacle is easily solvable and we take action quickly, the project might be able to keep its expected duration. However, if the issue is more troublesome than we expected, the project schedule might be delayed, resulting in a potential shift to the next project delivery turn. On the other hand, the opposite can also happen. If the project turns out to be perfectly executed and no major issues are encountered, it is possible that we end up finishing all the tasks before the expected end date.

We will be using the optimization task as our time balancing method. Since our premise of the task is already flexible, it will be feasible to adjust the number of applications and how in depth they will be optimized depending on the project

progress and time spent until then.

If our project progress goes hand in hand with the expected duration, the optimization task will be given the expected amount of hours stated in the planning. If we encounter some issues or difficulties that cause significant but non major delays, the optimization task will be given less time and will result in an optimization of less tasks or not as in depth. However, if we manage to finish the rest of the tasks before the expected duration, the task will be extended by working on more performance analysis and optimizations for the ported applications.

The tasks which have the most potential deviation in terms of execution time are the tasks which carry the port development of the applications, along with their testing. Since Rodinia Benchmark Suite has a significant amount of applications, we might run into potential delays if some applications have a higher complexity than we expected.

2.1.3 Considerations

Since the project methodology is based off some of the important aspects of the agile method, almost everything is up to change during the scheduled meetings. Even if no major obstacles are encountered, the planning and tasking could be modified on the fly to certain extend, or even create new tasks if we find it plausible.

2.2 Resources

2.2.1 Human resources

Even though this project is only managed by one person, we still need to specify the roles that person needs to act as throughout the course of the project.

Role	Responsibility
Project Manager	Makes sure the project is progressing adequately, and the tasks are being executed properly within the deadlines.
Developer	In charge of developing the software for the project, in this case the porting of the applications.
Public Relations (PR)	Stays in contact with the direct stakeholder (project director) and discusses the progress and the next aim with him.

Table 2.1: Human resources

2.2.2 Hardware resources

- Customized desktop computer (intel core i5 7500, 16GB RAM, GTX 1050ti graphics card, 256GB SSD and 2TB HDD)	Will be used for creating virtual environments for testing, developing the applications and writing most of the documentation.
- ZedBoard Zynq-7000 ARM/FPGA SoC Development Board - AXIOM Board	Boards with ARM processor and FPGA acceleration, will be used as the main target for developing the applications.

Table 2.2: Hardware resources

2.2.3 Software resources

Microsoft Windows 10 Professional Edition 64bits	Microsoft Office 2016	Adobe Acrobat Reader DC
Linux Ubuntu 16.04 LTS	Vivado	Vivado HLS
Nanos++	Mercurium	AutoVivado
Petalinux	QEMU	

Table 2.3: Software resources

2.3 Task description

Even though some of these tasks could be done in parallel, there is only one person executing the project so we need to sequentialize them. The tasks described below are already ordered by their execution.

2.3.1 Early project documentation

The initial documentation will be written in order to establish the importance, solidity and schedule of the project. An initial presentation will also be held at the end of this phase.

2.3.2 Virtual environment setup

Since testing from scratch in a physical board can be slow, we will be using a virtual environment for the first steps of the porting. That environment will consist of a boot generated by Petalinux tools and BSC's own Linux operative system installation with all the required libraries to run OmpSs applications. We will be using QEMU as the emulator.

2.3.3 Porting from OpenMP to OmpSs

Since the Rodinia Benchmark Suite applications use OpenMP library, they make use of parallel directives, which are not compatible with OmpSs device targeting. Because of that, we will have to first transform those parallel directives into task based parallelism, so that OmpSs can then target the code to an FPGA in a later step.

2.3.4 Tests bench creation

We will need a proper test bench in order to ensure the OpenMP to OmpSs porting has been done successfully. This task could have been done before the porting, but it makes sense to design a test bench for the applications once you already have some knowledge on them.

2.3.5 Virtual testing

We will be testing the OmpSs applications in the QEMU emulated environment using the previously created test bench. If any result from a ported application differs from the original application, we will need to go back to task 3 and then redo the testing once it appears to be fixed.

2.3.6 Adding FPGA targeting

We will add the required directives for OmpSs to send the portion of the code that needs to be accelerated to AutoVivado, so that it can create both Vivado and Vivado HLS projects and compile the bitstream for the FPGA afterwards.

2.3.7 Board Testing

This time using the physical board instead of the virtual environment, we will test the FPGA accelerated applications. Similar to the virtual testing, we will go back to task 6 if there is any mismatch with the results.

2.3.8 Optimization

After we are done testing, we will choose a certain number of applications that will be further optimized. The exact number of applications will depend on how the project is going compared to the expected duration.

We will make use of Vivado HLS project options and directives to optimize the integrated circuits that will be simulated on the FPGA. We will be using the instrumentation that comes with nanos++ to help us decide which is the best performing optimization for each chosen application.

2.3.9 Final documentation

When the project is finally developed, we will have to explain our results and conclusions to the final document. The project presentation will also be held sometime after the final document has been handled.

2.4 Estimated time

Task	Estimated time
Early project documentation	100 hours
Virtual environment setup	50 hours
Porting from OpenMP to OmpSs	90 hours
Test bench creation	40 hours
Virtual testing	20 hours
Adding FPGA targeting	10 hours
Board testing	40 hours
Optimization	100 hours
Final documentation	80 hours
Total	530 hours

Table 2.4: Estimated task time

2.5 Gantt chart

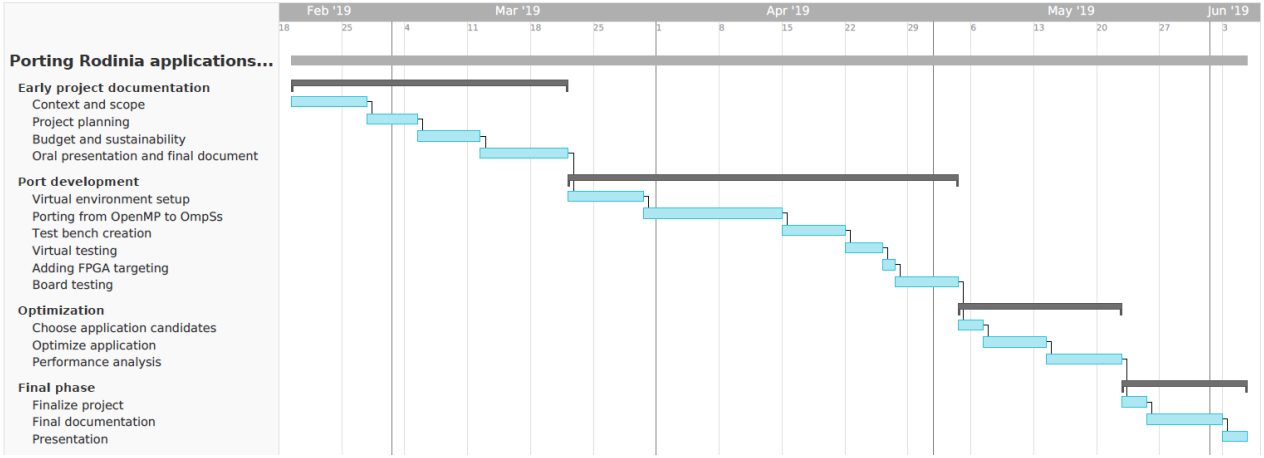


Figure 2.1: Gantt chart

Chapter 3

Budget

3.1 Initial documentation

3.1.1 Human resources

We are using market remuneration studies [8] as an approximate for the salary of each role which is needed to complete the project. We are using the study in the technological field to estimate the salaries for the Project Manager and the Developer, and the study in the commercial field for the estimation of the Public Relations role.

Role	Salary (Euro / Hour)	Work time(hours)	Estimated cost (Euro)
Project manager	20	100 hours	2000
Public Relations	13	5 hours	65
Total	-	-	2065

Table 3.1: Human budget for the initial documentation

In this case, we will only write documentation, so the developer role will not take part in it. The Public Relations role, while not taking part in any specific task defined in the project, is responsible for the communication between the main stakeholder and the rest of the team. In other words, his work time defines the amount of communication needed for every phase.

3.1.2 Hardware resources

For the budget to be accurate, we have to consider the amortization cost for the hardware and software used in the project. Furthermore, we are going to use the maximum annual coefficient of amortization stated by the government of Spain [9] for each case and convert it into the minimum time needed for amortization (in years). In this case, we get a coefficient of 25% for the desktop computer, which is equivalent to a minimum of 4 years of amortization. We are going to assume an average usage of 8 hours per day for the equipment used.

Name	Cost (Euro)	Coefficient(%)	Total hours	Amortization (Euro)
Desktop PC	1024.40	25%	100 hours	8.77
Total	-	-	-	8.77

Table 3.2: Hardware budget for the initial documentation

3.1.3 Software resources

We are going to use the same method as the hardware for calculating the amortization costs of the software used for the project. In this case, the coefficient for IT applications is 33%, which is equivalent to a minimum of 3 years needed for amortization. Just like before, we are going to assume an average usage of 8 hours per day for every software used.

Name	Cost (Euro)	Coefficient(%)	Total hours	Amortization (Euro)
Microsoft Windows 10 Pro	199.99	33%	100 hours	2.26
Microsoft Office 2016	439.99	33%	100 hours	4.97
Adobe Acrobat Reader DC	0	-	-	0
Total	-	-	-	7.23

Table 3.3: Software budget for the initial documentation

3.2 Porting and optimization phase

3.2.1 Human resources

Role	Salary (Euro / Hour)	Work time(hours)	Estimated cost (Euro)
Developer	13	350 hours	4550
Public Relations	13	10 hours	130
Total	-	-	4680

Table 3.4: Human budget for the porting and optimization phase

During the porting phase, the developer will do most of the job. Public Relations will stay in contact with the main stakeholder in a biweekly basis to decide what should be done next.

3.2.2 Hardware resources

In addition to the desktop computer, we will perform some of the tasks during the porting phase using the development boards. Both boards have the same coefficient as the desktop computer, and will only be used during the two final tasks of the porting phase (board testing and optimization).

Name	Cost (Euro)	Coefficient(%)	Total hours	Amortization (Euro)
Desktop PC	1024.40	25%	350 hours	30.70
Zedboard Zynq-7000 ARM/FPGA SoC Development Board	399.59	25%	140 hours	4.79
AXIOM Board	2000	25%	140 hours	23.97
Total	-	-	-	59.46

Table 3.5: Hardware budget for the porting and optimization phase

3.2.3 Software resources

During this phase, we will work in a Linux environment. We will be using Vivado and Vivado HLS from the Vivado Design Suite software, license of which is the floating type.

Name	Cost (Euro)	Coefficient(%)	Total hours	Amortization (Euro)
Linux Ubuntu 16.04 LTS	0	-	-	0
Vivado Design Suite	3595	33%	150 hours	60.94
Petalinux	0	-	-	0
Nanox++	0	-	-	0
Mercurium	0	-	-	0
AutoVivado	0	-	-	0
QEMU	0	-	-	0
Total	-	-	-	60.94

Table 3.6: Software budget for the porting and optimization phase

3.3 Final documentation

3.3.1 Human resources

Role	Salary (Euro / Hour)	Work time(hours)	Estimated cost (Euro)
Project manager	20	8 hours	1600
Public Relations	13	5 hours	65
Total	-	-	1665

Table 3.7: Human budget for the final documentation

3.3.2 Hardware resources

Name	Cost (Euro)	Coefficient(%)	Total hours	Amortization (Euro)
Desktop PC	1024.40	25%	80 hours	7.02
Total	-	-	-	7.02

Table 3.8: Hardware budget for the final documentation

3.3.3 Software resources

Name	Cost (Euro)	Coefficient(%)	Total hours	Amortization (Euro)
Microsoft Windows 10 Pro	199.99	33%	80 hours	1.81
Microsoft Office 2016	439.99	33%	80 hours	3.98
Adobe Acrobat Reader DC	0	-	-	0
Total	-	-	-	5.79

Table 3.9: Software budget for the final documentation

3.4 Indirect costs

3.4.1 Internet access

During the course of the project we will need internet access for a variety of reasons, such as information gathering, floating license access and communication with the project director. Because of that, it is only expected that we include such cost in our budget. The monthly internet access cost is 75€. Assuming 8 hours per day usage and a total amount of 530 hours, we end up with an estimated cost of 75*12*530/ (365*8) = 163.36€

3.4.2 Power consumption

In order to compute the cost of our power consumption throughout the course of the project, we searched information about the power consumption of the desktop computer and the development boards [10], along with the pricing of the kW per hour of our establishment [11].

Name	Power (Watts)	Cost (€/kWh)	Total hours	Estimated Cost (Euro)
Desktop PC	313W	0.145841	530 hours	24.19
Zedboard Zynq-7000 ARM/FPGA SoC Development Board	60W	0.145841	140 hours	1.23
AXIOM Board	15W	0.145841	140 hours	0.31
Total	-	-	-	25.72

Table 3.10: Power consumption budget

3.5 Unexpected costs

As with most projects, there is a chance that the cost is higher than expected due to unexpected events. In our case, the only significant budget deviation is the developer working more hours due to one or more applications being more complex to port than expected. More precisely, the deviation is likely to happen during the development of the ported applications along with their testing (tasks 3, 5, 6 and 7). If that is the case, the developer will increase his daily working hours to 10 and work during weekends if necessary.

Role	Salary (Euro / Hour)	Extra work time(hours)	Estimated cost (Euro)
Developer	13	50 hours	650
Total	-	-	650

Table 3.11: Unexpected costs

3.6 Total budget

Section	Human resources	Hardware resources	Software resources	Total cost(Euro)
Initial documentation	2065	8.77	7.23	2081
Porting phase	4940	59.46	60.94	4800.4
Final documentation	1665	7.02	5.79	1677.81
Internet access	-	-	-	163.36
Power consumption	-	-	-	25.72
Unexpected costs	-	-	-	650
Total	8410	75.25	73.96	9398.29

Table 3.12: Total budget

3.7 Control management

The amount of hours assigned to the creation of the initial documentation are enough to guarantee the completion of such task without any significant chance of failing to meet the time requirement.

As we mentioned previously, one of the main factors for budget deviations is the developer working more hours than the estimated amount, due to the unexpected complexity of one or more applications to be ported. We have already taken this into account when planning our budget.

However, deviations that surpass our expectations can still occur throughout the course of the development process. In order to manage and prevent these unexpected events as much as possible, we will save a registry of the real hours taken to perform each task. When we reach the optimization, the last task of the porting phase, we will check the total deviation from the estimated time.

If there is no substantial deviation, the last task will go as planned. However, if the deviation is significant we will spend more or less time on the optimization

task in order to make up for the time difference.

Regarding the final documentation, there is no telling how much time it will take since it will depend on the conclusions we arrive to and the amount of optimizations we are able to perform. However, we have set a time duration for the task which is unlikely to be surpassed unless major issues are encountered, which should not be happening this late in the project. If it does happen, our only solution is to make the project manager work more hours on the documentation. This, however, is so improbable that we have not included it in the unexpected costs.

Chapter 4

Sustainability report

4.1 Environmental

4.1.1 Project put into production

As we have mentioned in our planning, we will develop most of the project making use of a virtual environment, instead of using the targeted physical hardware from the very start. While this has a practical explanation, it also reduces the total amount of power we are going to use for the project. If we tested our applications using the physical boards, despite them having significantly less power consumption than the desktop computer, we would end up consuming more power because of the desktop computer staying on during the testing process regardless.

Taking in consideration the estimations for device power consumption and hours spent in the project used in the budget section, we end up with a total energy usage of 176 kWh.

4.1.2 Exploitation

As of today, there are not many ways to properly analyze the performance and efficiency of FPGA acceleration compared to other more traditional computing environments such as multi-core CPU and GPU.

By extending the Rodinia Benchmark Suite to FPGA environments, there can be an easier understanding on how this type of heterogeneous system performs in comparison to others in a specific domain of study. This will allow the potential improvement of current computations in terms of power efficiency, thus potentially reducing the ecological footprint.

4.2 Economic

We have estimated the project budget in the previous section, taking in consideration human, material and indirect resources.

Adding FPGA benchmarks to the equation will help companies and several domains of study evaluate with more possibilities the best economical hardware for their computing tasks. For example, depending on the algorithm size FPGAs that fit your design could be cheaper than other options and have similar performance.

4.3 Social

4.3.1 Project put into production

Working on the initial documentation of this project has taught me how important is to know as much as possible about the project you are doing before actually starting to develop it. It has also taught me that there needs to be a good reason for a project to exist, and you need to evaluate how solid your project idea is by thinking of how it can contribute to society environmentally, economically and socially speaking.

On the development side, this project will help me understand how FPGAs work, a device which has got my attention since the first time I was introduced to its concept. It will also help me improve my parallel programing skills along with the usage of instrumentation tools to evaluate parallel implementations.

4.3.2 Exploitation

As we mentioned previously, extending the Rodinia Benchmark Suite to FPGA based environments will ease the understanding of FPGA acceleration performance in comparison to other computation environments. Because of this, it is possible for some domains of study to improve their computation in a way that positively affects social aspects of our society.

4.4 Sustainability matrix

	Project put into production	Exploitation
Environmental	8	5
Economic	9	5
Social	10	3

Table 4.1: Sustainability matrix

Chapter 5

Follow-up Milestone

5.1 Work plan

As stated in the initial documentation, a virtual environment for the first development stage has been set up. However, due to a delay of the arrival date for the hardware boards and also on the interest of the project director, an extra workload was added to the task.

5.1.1 FPGA Simulation

In addition to the original work for the virtual environment setup, an FPGA simulation environment was developed. This allows programs compiled for FPGA based acceleration to be executed without the need for the actual hardware to be present. Although optimization is not feasible in a simulation environment, it is substantially useful to debug and make sure a program has the intended behavior and has a working FPGA tasking setup.

In the initial documentation, the intentions were to first develop the application ports in OmpSs using tasking with a generic device called Symmetric multi-processing (SMP) which utilizes several cores from a multi-core processor architecture. This would allow us to have an idea of how the FPGA tasks would behave, but would have most likely needed some extra tweaking before entering the FPGA hardware testing stage, since FPGA tasks have some limitations when it comes to data transfer. Thanks to the simulation environment, the tasks can already be tested as if they were being sent to an FPGA hardware, so most problems with the architecture can be detected without having to use the actual hardware. After the testing, the hardware will be finally used to perform the optimization.

The simulation environment is mainly composed of two parts: a slightly modified runtime and the simulation program. Nanos++ runtime has been slightly modified in order to adapt to the communication method of the simulation program. The simulation program is a program made in C language that executes tasks targeting the FPGA device, using the specified argument data, reading and/or modifying it depending on the specified FPGA computation. The program is also divided in two parts: the main generic code and the specific block code. The main code handles the communication with xdma/xtasks libraries (part of nanos++ runtime), while the block code has all the computation tasks and data structures needed for a specific program.

5.1.2 Impact of the changes

The additional workload took an approximate of 60 hours. However, since this work will ease and improve the early testing of the applications, it is expected that a portion of that work time will be saved in the hardware testing later on.

5.1.3 Stage reached

During the course of the project until today, the virtual environment for the first development stage has been done (along with the extra workload) and most of the early development has also been made, with a few applications left to develop. Note that the early development has been improved over what was planned on the initial documentation and it now allows FPGA task testing.

5.2 Follow-up conclusion

Due to the extra workload added to the project and delay of the hardware arrival, we have concluded that the project delivery should be postponed for the next delivery date, in October.

Chapter 6

Final Milestone

6.1 Objective and Scope

Throughout the course of the project, both the objective and the scope have stayed the same. However, it can be now stated that a total amount of 7 applications from the Rodinia Benchmark Suite have been ported to OmpSs@FPGA environment, 6 of which have been optimized to a certain degree.

6.2 Work plan

As specified in the follow-up milestone, the FPGA simulation environment was utilized to port the applications without the need for a physical device. Even though the development of the simulation environment added an extra workload to the project, the practicality of not having to use a physical board for the initial testing far outweighed its development time.

There have been a few changes since the last milestone regarding the hardware that was going to be used for the project. For one, the physical boards that were planned to be used were changed to a single FPGA device called Alpha Data. Unlike the previous hardware boards, this device is not dependent on an ARM host machine from which you have to execute the FPGA based applications. Instead, the FPGA device is connected through PCI-Express to a server from the BSC, on which the programs are compiled, the bitstreams are generated and the applications are finally executed. Because of that, Petalinux tools were no longer needed for the project.

Furthermore, since now the applications are first tested with the FPGA targeting already in place, the tasks 3 (Porting from OpenMP to OmpSs) and 6

(Adding FPGA targeting) are now part of the same task. The two respective testing phases have also merged into one.

6.3 Costs

Due to the changes in the hardware used for optimization, the inferred costs from the initial mileage are not accurate. A new budget has been calculated taking all these changes into consideration along with the real time spent on each phase. The budget for the initial documentation was accurate, and so did not need to be changed.

6.3.1 Porting and Optimization phase

More time for communication with the project director was needed due to the delay of the hardware availability.

Role	Salary (Euro / Hour)	Work time(hours)	Estimated cost (Euro)
Developer	13	350 hours	4550
Public Relations	13	30 hours	390
Total	-	-	4940

Table 6.1: Final human costs for the porting and optimization phase

Moreover, the porting part of the phase did not need the physical FPGA hardware thanks to the simulation environment. The time for the actual porting of the applications was substantially lower than what it was originally going to take, so the additional workload of the FPGA simulation environment creation fits in the original time schedule of the porting phase. The physical boards were changed to the Alpha Data device along with the BSC server that is connected to.

Name	Cost (Euro)	Coefficient(%)	Total hours	Amortization (Euro)
Desktop PC	1024.40	25%	350 hours	30.70
Alpha Data device	3310.00	25%	100 hours	28.34
BSC Server	4127.00	25%	100 hours	35.33
Total	-	-	-	94.37

Table 6.2: Final hardware costs for the porting and optimization phase

The fact that the physical hardware was not needed for testing the applications also reduced the usage time of the Vivado Design Suite.

Name	Cost (Euro)	Coefficient(%)	Total hours	Amortization (Euro)
Linux Ubuntu 16.04 LTS	0	-	-	0
Vivado Design Suite	3595	33%	100 hours	40.63
Petalinux	0	-	-	0
Nanox++	0	-	-	0
Mercurium	0	-	-	0
AutoVivado	0	-	-	0
QEMU	0	-	-	0
Total	-	-	-	40.63

Table 6.3: Final software costs for the porting and optimization phase

6.3.2 Final documentation

Instead of using Microsoft Word like on the initial documentation, it was decided that LaTeX was going to be used.

Name	Cost (Euro)	Coefficient(%)	Total hours	Amortization (Euro)
Microsoft Windows 10 Pro	199.99	33%	80 hours	1.81
LaTeX	0	-	-	0
Adobe Acrobat Reader DC	0	-	-	0
Total	-	-	-	1.81

Table 6.4: Final software costs for the final documentation phase

6.3.3 Power consumption

For the power consumption cost table, the two previous physical boards are swapped by the Alpha Data device and the BSC server. The power consumption of the alpha device was approximated by using the highest usage found in a benchmark sheet [12] by Xilinx regarding the Virtex-7 X690T FPGA, the FPGA model that the Alpha Data device uses. The power consumption from both the Alpha Data device and the server was estimated by the current average PVPC value in Spain [13].

Name	Power (Watts)	Cost (€/kWh)	Total hours	Estimated Cost (Euro)
Desktop PC	313W	0.145841	530 hours	24.19
Alpha Data device	26W	0.11574	100 hours	0.30
BSC server	120W	0.11574	100 hours	0.93
Total	-	-	-	25.88

Table 6.5: Final power consumption costs

6.3.4 Total final cost

In the initial documentation, an additional budget section was included as 'unexpected costs', in case the porting took more time than expected. However, thanks to the development of a simple FPGA simulation environment, the

porting phase finished in due time. Because of that, the section has been removed from the final budget.

Section	Human resources	Hardware resources	Software resources	Total cost(Euro)
Initial documentation	2065	8.77	7.23	2081
Porting phase	4940	94.37	40.63	5075
Final documentation	1665	7.02	1.81	1673.83
Internet access	-	-	-	163.36
Power consumption	-	-	-	25.88
Total	-	-	-	9019.07

Table 6.6: Total final costs

Chapter 7

Porting

In this chapter, we introduce and examine the applications to be ported to the FPGA environment, enumerating their characteristics and opportunities for optimization.

7.1 BFS

7.1.1 Description

BFS is the common Breadth-first Search graph traversing algorithm used in a wide variety of areas. It consists of searching for all the nodes of the same depth before starting to search the nodes of a higher depth. There are 2 main procedures that can be parallelized:

Procedure 1- Traversal of nodes of a certain depth through vertices in search of other nodes which are not visited yet. Updates the cost of each one of these nodes.

Procedure 2- Updates the tree structures for the graph traversal of the next depth. If it doesn't find any node to be updated the algorithm is completed, otherwise it prepares the structures for the next iteration.

7.1.2 Changes to the original application

Since the original parallelism was done using the 'parallel for' syntax from OpenMP library, it was changed to an OmpSs task, along with the manual creation of iteration blocks that every task will be assigned to.

Furthermore, the second procedure was ported for FPGA acceleration, due to the accesses on the structures always being to predictable positions. Since the FPGA can only receive fixed amounts of data, the last chunk of iterations that doesn't fill an entire block (if it's the case) is computed in the main thread.

The first procedure, however, requires arbitrary access to some of the structures in a non-predictable way. Because of that, it's not possible to bring only a part of these structures to the FPGA, but rather needs the whole structures to ensure all accesses are done correctly. At the same time, the size of these structures are heavily dependent on the amount of edges/nodes the graph has in every use case. Because of all these reasons, it was decided that the first procedure would be left as an SMP task (multi-core processor task).

7.2 NN

7.2.1 Description

NN (Nearest Neighbors), as the name suggests, finds the k-nearest neighbors in an unstructured data set. For each record in the data set, it uses the latitude and longitude to compute the distance between the record and the target [14]. The process of computing the distance between the target and the records can be parallelized.

7.2.2 Changes to the original application

As always, the 'parallel for' syntax was changed to an OmpSs task. Furthermore, an iteration block is defined for every task created. The k-nearest neighbors procedure was made an FPGA task successfully, since its parameters are located in a predictable way. The last chunk of iterations that doesn't fill an entire block (if it's the case) is computed in the main thread.

The original application used the parsing function 'atof' inside the parallel section, since the input data is initially given as a character array. Since Vivado HLS does not provide support for this specific function, a simple implementation of 'atof' was made.

7.3 Pathfinder

7.3.1 Description

Pathfinder uses dynamic programming to find a path on a 2d grid from the bottom row to the top row with the smallest accumulated weights, where each step of the path moves in a straight line or in diagonal [15]. One procedure can be parallelized, which is the iteration of an entire row with the smallest weights.

7.3.2 Changes to the original application

As always, the 'parallel for' syntax was changed to an OmpSs task. The task was also converted for FPGA acceleration. Since each iteration checks for both n-1 and n+1 positions in the 'src' structure, the first and last steps were excluded from the FPGA task, since it would become an issue for the FPGA's static size data (since these two lack one of the boundaries). The incomplete block of iterations at the end is also taken in consideration.

7.4 NW

7.4.1 Description

NW (Needleman-Wunsch) is a global optimization method for DNA sequence alignments. The potential pairs of sequences are organized in a 2d matrix. In the first step, the algorithm fills the matrix from top left to bottom right, step-by-step. The optimum alignment is the pathway through the array with maximum score, where the score is the value of the maximum weighted path ending at that cell. Thus, the value of each data element depends on the values of its northwest-, north- and west-adjacent elements. In the second step, the maximum path is traced backward to deduce the optimal alignment [16]. There are 4 procedures that can be parallelized:

Procedure 1- Copying global structure 'referrence' to local memory.
Procedure 2- Copying global structure 'input_itemset' to local memory.
Procedure 3- Computation of the main algorithm.
Procedure 4- Copy results to global memory.

7.4.2 Changes to the original application

As always, the 'parallel for' syntax was changed to an OmpSs task. There is no need to add block creation, since the original application already implemented

blocks. Procedures 1,2 and 4 only consist of assigning global array values to a local copy or vice versa, so there is no point in converting those into FPGA tasks. For that reason, only the third procedure was targeted for an FPGA device while the rest were converted into SMP tasks. There is no waiting between the 2 first procedures, since both can be done in parallel.

In this case, since the input was originally forced to be multiple of 16, there is no need to consider the case were there is an incomplete block of iterations at the end.

7.5 Hotspot

7.5.1 Description

Hotspot is a widely used tool to estimate processor temperature based on architectural floorplan and simulated power measurements. The thermal simulation iteratively solves a series of differential equations for block. Each output cell in the computational grid represents the average temperature value of the corresponding area of the chip [17]. The entire computation can be parallelized.

7.5.2 Changes to the original application

As always, the 'parallel for' syntax was changed to an OmpSs task, along with dividing the computation in several blocks. Each task has a certain number of rows to compute, and have no data dependences between each other. Since the algorithm requires accessing the previous and the next row of the coordinate being computed, a low and high boundary have been added to every FPGA task. However, such array is never written, which means it doesn't add any extra data dependencies and all FPGA tasks can potentially run at the same time (if the hardware allows it).

7.6 SRAD

7.6.1 Description

SRAD (Speckle Reducing Anisotropic Diffusion) is a diffusion method for ultrasonic and radar imaging applications based on partial differential equations. It is used to remove locally correlated noise, known as speckles, without destroying important image features [18]. There are two computations blocks

which can be parallelized:

Procedure 1- Directional derivatives, ICOV and diffusion coefficient.
Procedure 2- Diverge and image update.

7.6.2 Changes to the original application

As always, the 'parallel for' syntax was changed to an OmpSs task, along with dividing the computation in several blocks. Each task computes a certain number of rows, and have no data dependences with each other. Since the first procedure needs to access the previous and the next row of the input image, low and high boundaries have been added to the tasks. The second procedure needs access to the next row of the input matrix, so only the high boundary has been added. Neither of these inputs are modified while the tasks are running, and thus don't create any data dependences.

7.7 Myocyte

7.7.1 Description

The application models cardiac myocyte (heart muscle cell) and simulates its behavior according to the work by Saucerman and Bers. The model integrates cardiac myocyte electrical activity with the calcineurin pathway, which is a key aspect of the development of heart failure. It can be used to identify potential therapeutic targets that may be useful for the treatment of heart failure [19].

Biochemical reactions, ion transport and electrical activity in the cell are modeled with 91 ordinary differential equations (ODEs) that are determined by more than 200 experimentally validated parameters [20]. There are 2 procedures than can be parallelized:

Procedure 1- ECC(Excitation-Contraction Coupling) simulation.
Procedure 2- CaM(Calmodulin) simulation.

7.7.2 Changes to the original application

As always, the 'parallel for' syntax was changed to an OmpSs task. Thanks to ECC and CaM functions needing a static amount of parameters, both were successfully converted to FPGA tasks.

Both functions were also changed slightly in the way the structures were received. Instead of the structures being sent on its entirety and then using the offset to know where to start accessing them, the pointer was removed from the function header and instead the structures are sent with the offset already computed. That way you don't need to send the entirety of the structures, but rather the block that is needed for each computation.

The CaM function had a return value, so it was changed to an output parameter because the OmpSs library only supports void type task functions.

Chapter 8

Optimizations

8.1 Considerations and limitations

There are a few considerations and limitations which can affect the optimization process of the selected applications, and as such they need to be specified. The most obvious but very important limitation is the time constraint. Since every bitstream generation process takes from at least 3 hours to an indefinite amount of hours (depending on how complex the application and its Vivado HLS optimizations are), the amount of testing and 'trial and error' will be significantly lower compared to the standard optimization procedure for CPU and GPU based applications.

If the bitstream generation of a specific optimization takes an unreasonable amount of time (12+ hours), it will be considered stuck and the optimization will be discarded due to previously specified time limitations. Since the FPGA device resources are also limited, any optimization that exceeds the usage of any of its resources will not be put in practice and a less resource hungry optimization will be applied instead.

Once the applications are optimized, execution traces will be generated by the Extrae tool and they will be analyzed and explained using the Paraver trace viewer software.

8.2 Transformations

Some applications will be changed in a specific manner that only makes sense for them, but there are a few generic procedures that most applications will have in common.

8.2.1 Increased size of data input

As the title implies, the size of the input data can be increased so that the FPGA block can treat a bigger chunk of data at a time. This is very important since the one of biggest bottlenecks of FPGA heterogeneous computing is the data transfer and communication between the host and the FPGA.

While you are still going to send the same amount of final input data throughout the course of the application, sending very small chunks of data can result in a substantial communication overhead between the two devices, making the application slower. However, on the other hand, you might run into BRAM capacity issues if you make the size too big, especially if you are dealing with 2-dimension arrays. The input data size will be usually referred as block size or 'BSIZE'.

8.2.2 Increased number of IP core instances

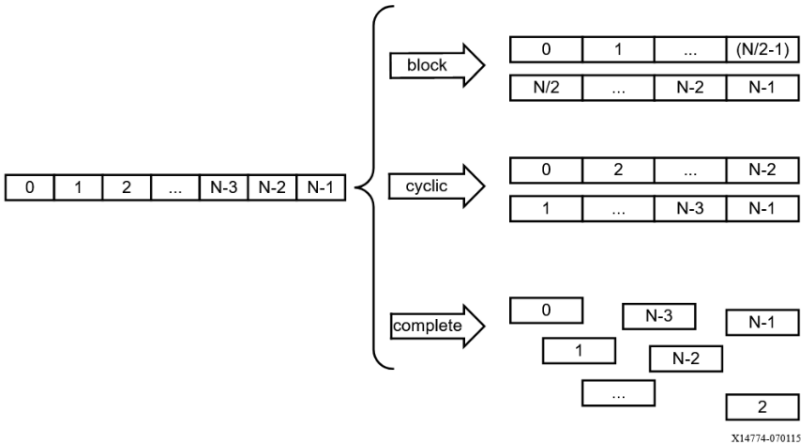
If the number of instances of each FPGA task is not specified, AutoVivado will tell the synthesizer to create a single IP core for each one of them. However, since IP cores can be executed in parallel with each other, a lot of computation work will benefit from having two or more instances of the same task. This method can reach higher performance than just increasing the task bandwidth, but will also duplicate every FPGA resource as many times as instances you choose to have.

8.2.3 Partition input data arrays

Input data arrays can be partitioned into smaller arrays for a variety of reasons. Arrays in the FPGA are implemented as single memory resources with a limited number of read/write ports. By splitting those into smaller arrays, you can effectively increase the number of read/write ports you have in total. This functionality can significantly improve the performance of other Vivado HLS optimizations like loop pipelining or loop unrolling [21].

There are 3 types of array partitioning: block, cyclic and complete. Block partitioning consists of simply dividing the original array in as many arrays as the specified factor in a consecutive manner, while cyclic partitioning allows you to spread consecutive elements of the array into different smaller arrays. Complete partitioning makes each element its own entity, similarly to registers. Note that complete partitioning does not use the factor parameter. Figure 8.1

shows an example of array partitioning with factor 2.



[22]

Figure 8.1: Example of array partitioning with Vivado HLS tools

8.2.4 Pipelining

The concept of pipelining consists of dividing a certain process in several phases so that you end up with a performance of one process per cycle (after a certain amount of initial delay). The HLS loop pipeline feature uses that method to optimize the loop body or function in such way that you achieve an iteration done every x cycles. Its performance can be greatly increased by partitioning the input data used inside the loop, as previously mentioned [23].

This optimization can significantly increase the synthesis process duration, or fail to achieve the desired result if the loop body is too complex. It also occupies a substantial amount of resources depending on how large the section to be pipelined is.

8.3 BFS

8.3.1 Use case analysis

For this application, the input data consists of a graph. Specifically, it contains the number of nodes, the connections between them, the initial node, the number of

edges and the cost of each one. The default use case that the application had was fairly simple, so we generated a bigger graph using the already provided 'inputGen' tool in the Rodinia Suite.

8.3.2 State of the non-optimized application

The first intention was to use boolean arrays as the FPGA block input data. However, due to for now unknown circumstances, the synthesis process failed and the approach couldn't be implemented. The first solution to this issue was to simply replace the boolean arrays with integer arrays, which meant using 4 bytes for every boolean. Due to this highly inefficient usage of bandwidth, the first working application was extremely slow, to the point of not being usable.

The average duration of the application at this initial stage was of 16.25 seconds.

8.3.3 Bandwidth optimization

The first thing that needed to be changed was the way the data was transferred to the FPGA block. The arrays were left as integer type, however instead of each one of them having a single boolean worth of data, every integer contained a total of 32 booleans, one for each bit. This way, while adding slightly more complex read/write operations, it significantly improved our bandwidth efficiency. Both operations are shown in Code 8.1.

```
1
2  inline unsigned int getVal(unsigned int *array, int pos){
3    return ((array[pos/(sizeof(unsigned int)*8)] >> ((sizeof(unsigned int)*8) - pos%(sizeof(
       unsigned int)*8) - 1)) & 0x1);
4  }
5
6  inline void setVal(unsigned int *array, int pos, bool val){
7    if(!val) {
8      array[pos/(sizeof(unsigned int)*8)] &= ((1 << ((sizeof(unsigned int)*8) - pos%(sizeof(
       unsigned int)*8) - 1)) ^ 0xFFFFFFFF);
9    }
10   else {
11     array[pos/(sizeof(unsigned int)*8)] |= (1 << ((sizeof(unsigned int)*8) - pos%(sizeof(
       unsigned int)*8) - 1));
12   }
13 }
```

Code 8.1: Read and write operations for packed boolean arrays

At this point, the average duration of the application was of 3.33 seconds, a very substantial improvement.

8.3.4 SMP task optimization

The bandwidth optimization had a very positive impact on the efficiency regarding the FPGA task, however it forced the SMP task of the application to be serialized due to random accesses of the array 'h_updating_graph_mask'. Packing 32 booleans in a single integer variable added the possibility of 2 or more tasks accessing at the same integer through different booleans. Since accessing and writing the individual booleans require several operations on the destined integer, a race condition can appear with ease.

In order to avoid the full serialization of this task, atomic clauses were added to the specific writing of this array (only on that specific case) in order to ensure that the addresses were never accessed simultaneously.The atomic version of the write operation is shown in Code 8.2.

```
1
2  inline void setValAtomic(unsigned int *array, int pos, bool val){
3    if(!val) {
4      #pragma omp atomic
5      array[pos/(sizeof(unsigned int)*8)] &= ((1 << ((sizeof(unsigned int)*8) - pos%(sizeof(
       unsigned int)*8) - 1)) ^ 0xFFFFFFFF);
6    }
7    else {
8      #pragma omp atomic
9      array[pos/(sizeof(unsigned int)*8)] |= (1 << ((sizeof(unsigned int)*8) - pos%(sizeof(
       unsigned int)*8) - 1));
10   }
11 }
```

Code 8.2: Atomic version of the write operation

After this change, the average duration of the application was of 2.73 seconds, a not great but still reasonable improvement.

8.3.5 Adding extra instances

Since the FPGA block for this application is really simple and does not consume a lot of resources, it is worth adding a few extra instances that will be executed in parallel to gain more efficiency. Specifically, the total amount of instances was changed to 4.

The average duration of the application at this point was of 1.37 seconds.

8.3.6 Optimization overview

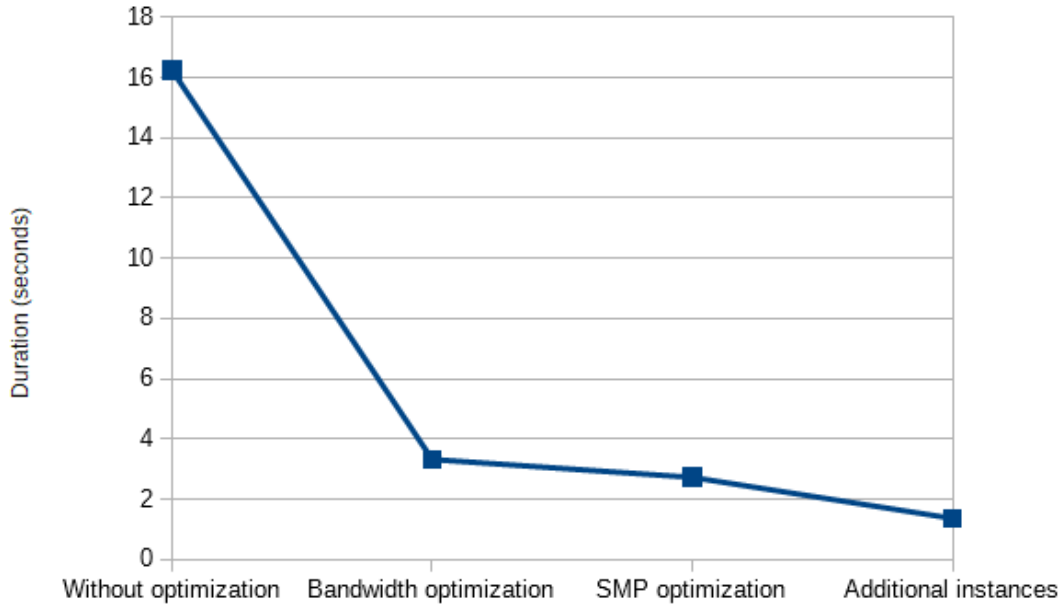


Figure 8.2: BFS optimization chart

The total performance gain from the accumulated optimizations of the BFS application (based on the non-optimized version) is 11.86.

8.3.7 Analysis

A portion of the trace from the non-optimized bfs application is shown in Figure 8.3.

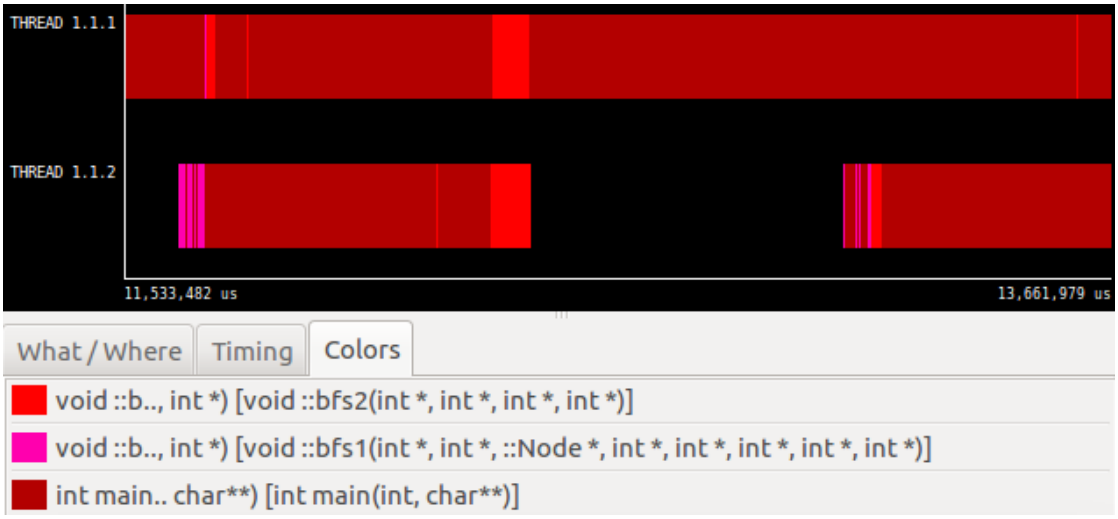


Figure 8.3: BFS execution trace without optimizations

We can observe that the application barely manages to complete an iteration and a half of the main loop during that interval. While the main function seems to be doing all the job in the first half of the timeline, some bfs2 FPGA tasks are hidden in the trace. If the trace is zoomed in by that section we can see more clearly what is happening (shown in Figure 8.4).

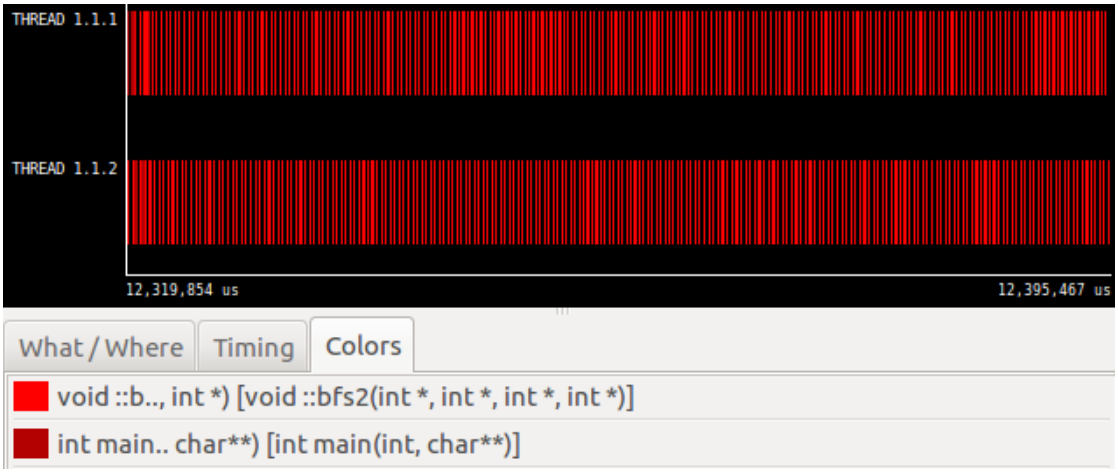


Figure 8.4: Zoomed in BFS execution trace without optimizations

It seems that both threads are alternating the preparation work before creating the task and the execution of the tasks themselves.

Looking again at the first image, we can also observe that there is a significantly large computation being done by the main function between the end of the bfs2 FPGA tasks and the start of the bfs1 SMP tasks from the next loop iteration. This could represent the reduction process of the variable 'stop' that is executed by the main function after all the FPGA tasks have been completed. Since the bandwidth size is only 1024 and each integer represents a single boolean value, the main function has to apply a very large reduction.

The trace from the application with the bandwidth optimization is shown in Figure 8.5.

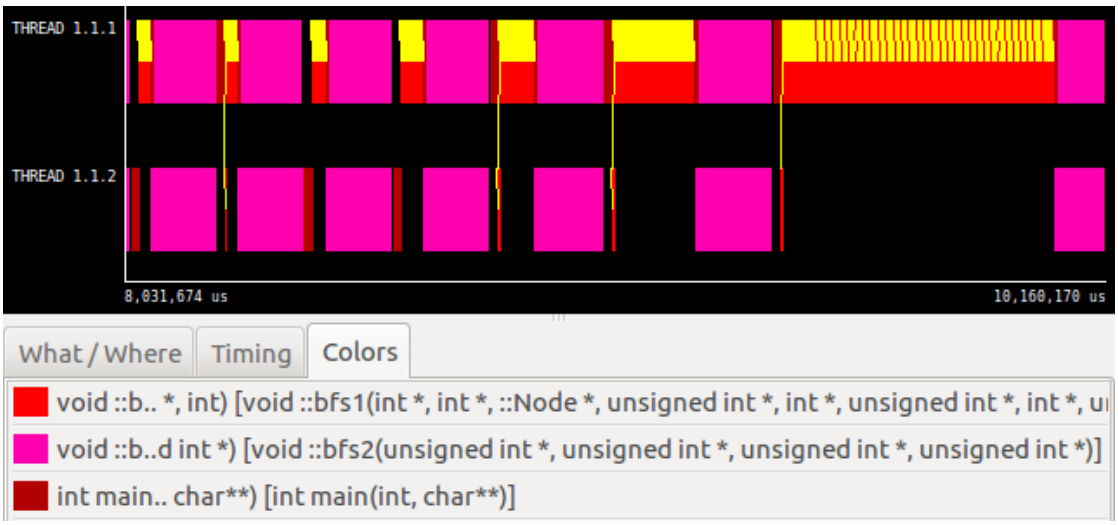


Figure 8.5: BFS execution trace with bandwidth optimization

Both the time to execute all the bfs2 FPGA tasks and the reduction of the variable 'stop' after all of them are finished have been reduced to a significant degree. However, the packing of some of the structures has caused a false dependence between bfs1 tasks, due to the random access on one of the packed variables which could cause 2 different tasks to modify the same integer at the same time. Because of that, the dependence has serialized the execution of the bfs1 tasks (note the yellow lines).

The trace from the application with SMP task optimization is shown in Figure 8.6.

Figure 8.6: BFS execution trace with SMP optimization

We immediately see that the dependence lines are gone, and the average duration of the bfs1 task groups is slightly reduced. While the tasks are no longer done sequentially, some overhead might have been produced due to the atomic write operation (Figure 8.2).

The final optimization, adding additional instances to the FPGA task, resulted in a substantially faster execution (shown in Figure 8.7).

Figure 8.7: BFS execution trace with additional instances

We can see that the bfs2 FPGA tasks are completed faster thanks to being able to send more than one of them concurrently. This can be seen more clearly if we zoom in on the FPGA task creation section.

The application with a single instance is shown in Figure 8.8, and the application with 4 instances is shown Figure 8.9. We can see in the latter trace that the amount of instances in the same amount of time has almost quadrupled.

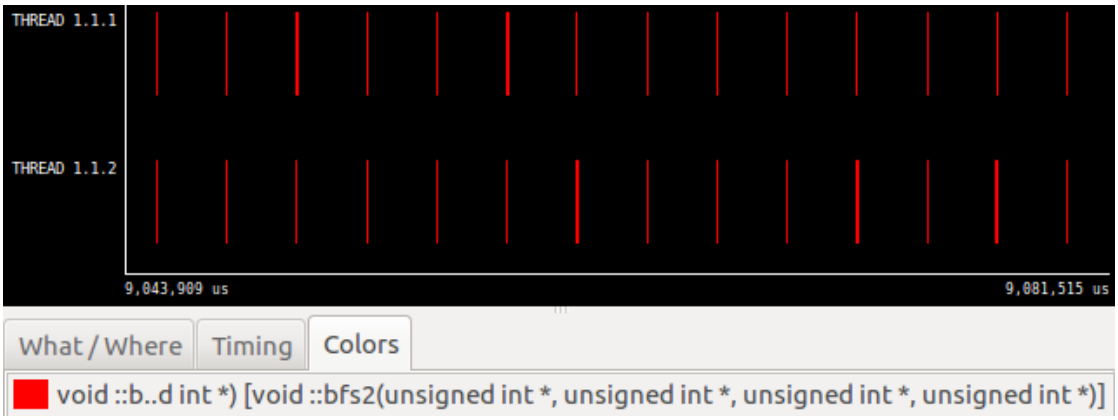


Figure 8.8: Zoomed in BFS execution trace with SMP optimization

Figure 8.9: Zoomed in BFS execution trace with additional instances

8.3.8 Conclusion

This BFS implementation, as observed, is certainly portable to an FPGA heterogeneous computing environment. However, there are a few limitations

when it comes to improving its performance.

For one, the fact that one of the tasks could not be easily implemented as an FPGA task (due to a random access problem, explained in section 7.1) already caps the total amount of optimization we can achieve using the FPGA device and the total workload that it manages to take. Secondly, while the booleans being packed in integers had a really solid performance increase at the start, it might have made other optimizations like pipelining and unrolling harder for the synthesizer to implement.

In conclusion, for this application to perform optimally in an FPGA heterogeneous computing environment, it needs not only an FPGA device but also a good performing host processor.

8.4 NN

8.4.1 Use case analysis

The input data in this application consists of a large amount of entries, including several information like year, month, day, hour, name, longitude and latitude. Specifically, these use case entries represent hurricanes with their arrival time at certain locations. Even though there is a lot of information about each entry, the algorithm only uses the longitude and latitude in order to compute the nearest neighbor algorithm.

The original use case for the application was really short and would not be suitable for optimization purposes, so a bigger use case was generated by the already provided 'inputGen' tool in the Rodinia Suite.

8.4.2 State of the non-optimized application

The original algorithm was given the entire information in characters, and extracted the longitude and latitude from them using the parsing function 'atof' from the 'stdlib' library which converted the specific characters into float type. Vivado HLS does not support this function, so the first idea was to simply make a simplified implementation of that parsing function and call it inside the FPGA block.

However, due to FPGA logic not being able to handle the parsing workload well, the first working version of the application was so slow it could not finish

the execution of the big use case with a reasonable amount of time. This could be because parsing is composed of mostly sequential iterations of a single address which has a limited amount of read ports, while also writing the result on a variable which has a data dependence with itself between iterations.

8.4.3 Data handling optimization

Since the FPGA logic did not seem to handle the parsing properly, that part of the data handling was moved out of the FPGA and placed in the host before starting to send the tasks. This way, not only the FPGA did not need to treat the input before computing ,but the needs of bandwidth were also significantly reduced due to only needing to send two float types instead of the entire information of each entry.

After this change, the duration of the application was of 13.35 seconds.

8.4.4 Adding extra instances

Once the parsing was out of FPGA block, 2 additional instances were added to the logic, upping the number of instances to a total of 3. The duration of the application at this stage was of 9.94 seconds, not a big improvement but certainly not small either.

8.4.5 Block size increase

The block size of the application was originally 1024. However, since the input data consists of 3 single dimension arrays, that number could be increased. The block size was increased to 8192, since increasing the input size further did not seem to improve the application's performance.

The duration of the application at that point was of 8.62 seconds.

8.4.6 Loop pipelining

The method that was chosen for pipelining is shown in Code 8.3. The original 'for' loop has been blocked, resulting in loops 'ii' and 'i'. The reason behind this is that you are able to pipeline based on the inner loop with a certain amount of iterations equal to the block factor. This means that Vivado HLS will only try to pipeline this many iterations at once. In this case, the block factor was decided to be 32.

```
1
2  void nn_block(float *tmp_lats, float *tmp_longs, float *target_lat, float *target_long, float *
       z){
3    #pragma HLS ARRAY_PARTITION variable=tmp_lats cyclic factor=BLOCK_FACTOR
4    #pragma HLS ARRAY_PARTITION variable=tmp_longs cyclic factor=BLOCK_FACTOR
5    #pragma HLS ARRAY_PARTITION variable=z cyclic factor=BLOCK_FACTOR
6    for(int ii = 0; ii < BSIZE; ii += BLOCK_FACTOR){
7      #pragma HLS PIPELINE II=1
8      for (int i = 0; i < BLOCK_FACTOR; i++){
9        z[i+ii] = sqrt(( (tmp_lats[i+ii] - (*target_lat)) * (tmp_lats[i+ii] - (*target_lat)) ) +
       ( (tmp_longs[i+ii] - (*target_long)) * (tmp_longs[i+ii] - (*target_long)) ));
10         }
11   }
12 }
```

Code 8.3: Pipelined version of the NN FPGA task

The original 'for' loop has been blocked, resulting in loops 'ii' and 'i'. The reason behind this is that you are able to pipeline based on the inner loop with a certain amount of iterations equal to the block factor. This means that Vivado HLS will only try to pipeline this many iterations at once. In this case, the block factor was decided to be 32.

In order to avoid the encounter of read/write bottlenecks within the same array while pipelining, an array partition needs to be done in a cyclic way with the same factor that the reconstructed loop uses. This way consecutive iterations will access different array entities, making it much easier for the synthesizer to improve the performance of the application. After applying the changes, the duration of the application was of 7.15 seconds.

8.4.7 Optimization overview

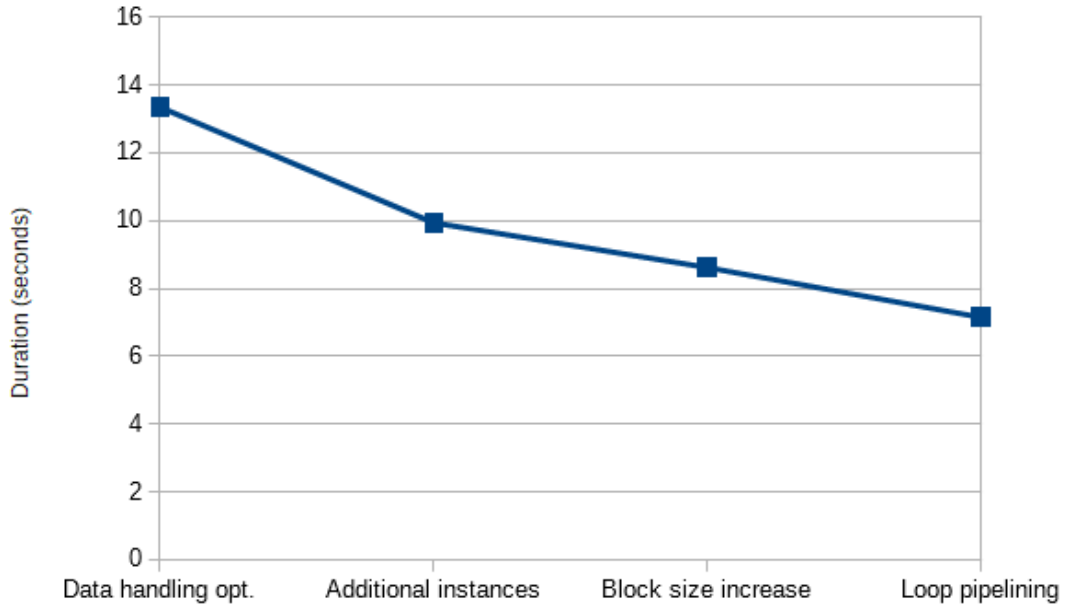


Figure 8.10: NN optimization chart

The total performance gain from the accumulated optimizations of the NN application (based on the data-handling optimization version) is 1.87.

8.4.8 Analysis

The first working version of the application that finished in a reasonable time (shown in Figure 8.11) consisted on reading and treating the float values before sending them to the FPGA task.

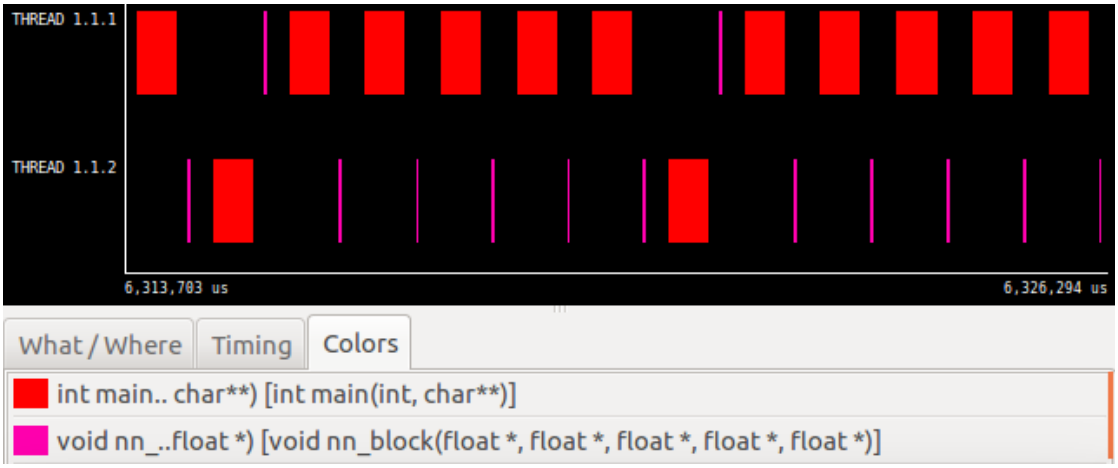


Figure 8.11: NN execution trace with without optimizations

You can observe the read bursts along with the float value parses outside of the FPGA function, creating a single task after that. There is also the loop that takes care of updating the k-nearest neighbors array of the previous iteration.

The improved version with additional instances is shown in Figure 8.12.

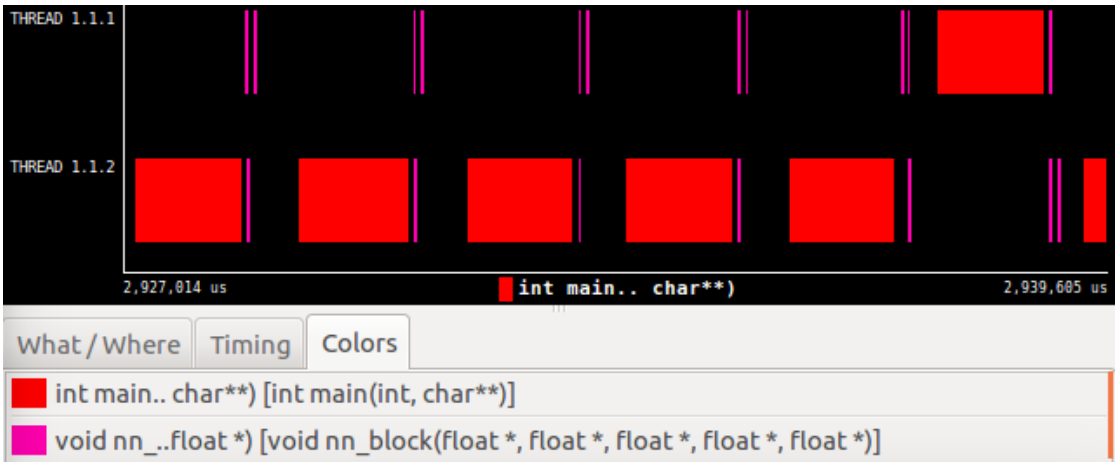


Figure 8.12: NN execution trace with additional instances

Instead of reading and treating exactly for the bandwidth of one FPGA task like the previous version, it does that for the total amount of instances which the FPGA block has. You can see the read/parse computational workload being tripled, along with the 3 tasks being created afterwards. While not being a really big improvement, it certainly improves the overall performance without margin

of error.

The next optimization consisted on increasing the block size, shown in Figure 8.13.

Figure 8.13: NN execution trace with increased block size

While the number of FPGA tasks has been substantially reduced and the workload of the main thread seems bigger, since the block size increase has made the bandwidth several times bigger the overall performance has increased slightly.

The last version of the application improves the performance of the FPGA block itself by applying pipelining to the inner loop. The partial trace is shown in Figure 8.14.

Figure 8.14: NN execution trace with pipelining

You can see the next read burst starting sooner thanks to the FPGA block taking less time to finish. Like in the previous case, it is not a big performance increase but still definitely noticeable.

8.4.9 Conclusion

Once taken care of the 'atof' problem we had at first, the application can certainly be ported to an FPGA heterogeneous environment with acceptable performance. However, there are still some limitations that prevent further FPGA optimizations to achieve higher overall performance increase.

The first one is the loop that takes care of updating the entries from the current k-nearest neighbors with the newly processed array. Since all the iterations of the loop can potentially modify the same array, the process cannot be parallelized and thus it has to be executed sequentially.

Secondly, the parsing of the float values from the character array also takes a significant amount of processing. An option could be to make it an SMP task, so that it can be parallelized in the host device. However, I think the best solution to this would be to have the input generator (either use case generator or real life data) already print the values with float representation. This would not only reduce a lot of work from the algorithm itself, but would barely increase the workload for the input generator, if not decrease it depending on how the data is originally taken.

In conclusion, for this application to perform optimally in an FPGA heterogeneous computing environment, it needs both an FPGA device and a good performing host processor.

8.5 Pathfinder

8.5.1 Use case analysis

This application does not require external data to be given as input, but rather the application itself already generates a randomized matrix (with a fixed seed, so every execution has the same effect) that represents a 2d space which will be used to find the path with the least total cost.

The default use case was determined to be good enough for optimization purposes, so there was not a need to change it.

8.5.2 State of the non-optimized application

Due to the fact that each iteration could potentially access the previous and next position of the array 'src', the first thought was to give these extra elements to the FPGA block by transferring BSIZE+2 elements of the array at a time. However, because of how the nanos runtime handles memory regions, this implementation did not work (as it overlapped memory regions).

To work around that problem, a slightly different approach was taken. Instead of giving BSIZE+2 elements of the 'src' array, the usual BSIZE elements were given along with 2 additional parameters, one for each border. In order to avoid memory region overlapping, these additional parameters were copied from the original array to a different one at the moment before calling the FPGA task.

The duration of the application at this initial stage was of 10.7 seconds.

8.5.3 Block size increase

Because of the original block size being very small and the data being one dimension only, the application had its block size increased to a total of 8192. That specific number was decided because of the application not improving much after trying to increase it further.

The duration of the application after this change was of 1.12 seconds, a significant increase over the original application. It also proves the application was having serious communication overheads due to the small block size.

8.5.4 Loop pipelining

The same pipelining strategy as the previous application was applied, with the same block factor. After applying the pipelining process, the duration of the application was reduced to 0.83 seconds.

8.5.5 Optimization overview

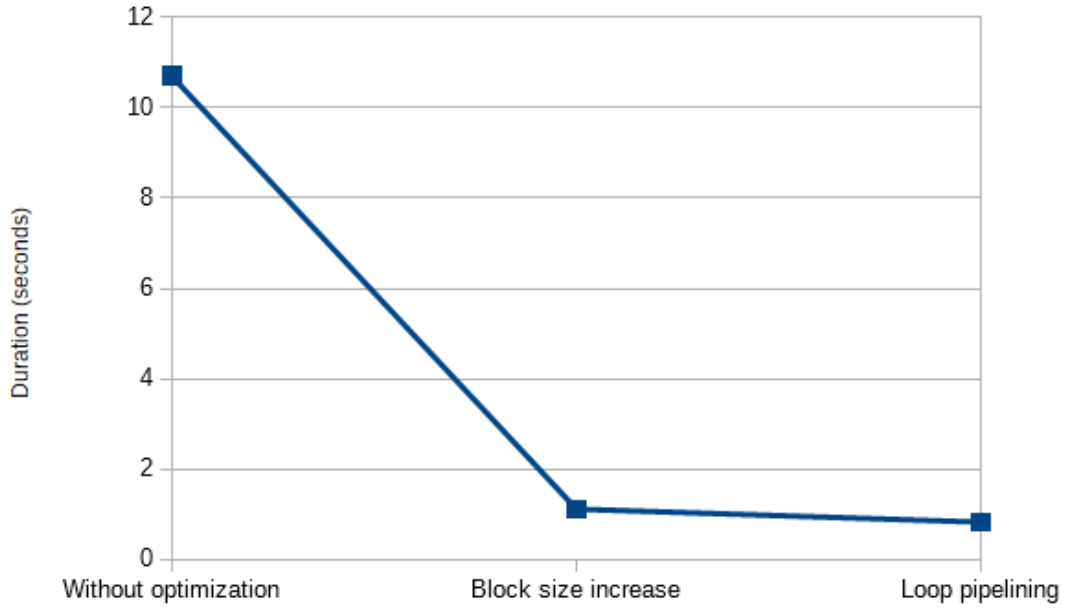


Figure 8.15: Pathfinder optimization chart

The total performance gain from the accumulated optimizations of the Pathfinder application (based on the non-optimized version) is 12.89.

8.5.6 Analysis

A portion of the trace for the non-optimized application is shown in Figure 8.16.

Figure 8.16: Pathfinder execution trace without optimizations

We can see that this portion of time can only handle a single iteration of the main application loop. Even though the iterate_n_block section of the trace looks like a contiguous line, it is actually composed by a lot of small tasks, as seen in Figure 8.17.

Figure 8.17: Zoomed in Pathfinder execution trace without optimizations

This shows a clear case of overhead due to the FPGA task bandwidth being too small compared to the total size of the use case input. Looking at the version with a much bigger block size, we can clearly see the application is benefiting from it (shown in Figure 8.18).

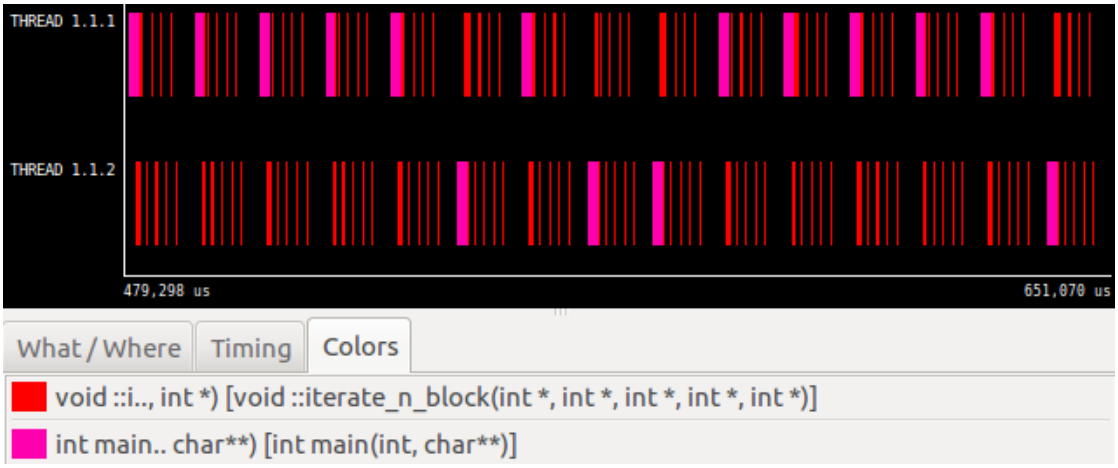


Figure 8.18: Pathfinder execution trace with increased block size

For each iteration of the non-optimized version, approximately 15 fit when the block size has been substantially increased.

The pipelined version reduces the execution time of the FPGA block, thus improving a bit more the overall performance of the application.

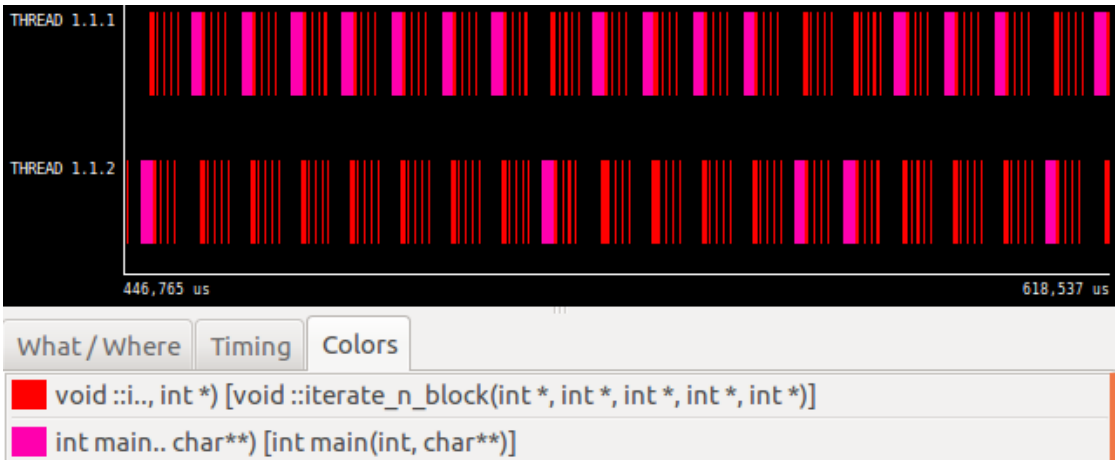


Figure 8.19: Pathfinder execution trace with pipelining

8.5.7 Conclusion

The application can most definitely be ported for FPGA heterogeneous computing. Not only that, but most of its workload can be computed inside the FPGA block, which gives more room when it comes to choosing a host that

would perform adequately for the application. At the start, it might seem that the application does a lot of work by generating the use case, however this process is only for testing purposes and you would have an input data file instead for real life applications.

In conclusion, the application can perform well in an FPGA heterogeneous computing environment without the host processor being the limiting factor of its overall performance.

8.6 NW

8.6.1 Use case analysis

The application does not need external input data as use case, but rather generates a square matrix representing a DNA sequence based on a fixed random seed and the columns/rows specified in the execution parameters.

The original amount of columns/rows of the default application execution script was a too small for optimization purposes, so a significantly bigger value was used instead.

8.6.2 State of the non-optimized application

The first working stage of the application used a block size of 32. While this value seems very small, it was actually a reasonable size to start with. The reason behind this is that the application's FPGA block handles two dimensions of that size, ending up with a much higher total bandwidth.

For this initial stage, the duration of the application was of 17.83 seconds.

8.6.3 Block size increase

While the starting block size was acceptable, it was tweaked to reach considerably higher bandwidth. Specifically, it was increased from 32 to 512. Reason for stopping at 512 was the fact that the FPGA did not have enough resources to handle a higher power of 2.

A block size of 256 elements with a total amount of 5 FPGA block instances was also taken in consideration, but was discarded due to having worse

performance than simply increasing the block size to 512 elements.

At this point, the duration of the application was of 8.30 seconds, a significant improvement over the initial stage.

8.6.4 Optimization overview

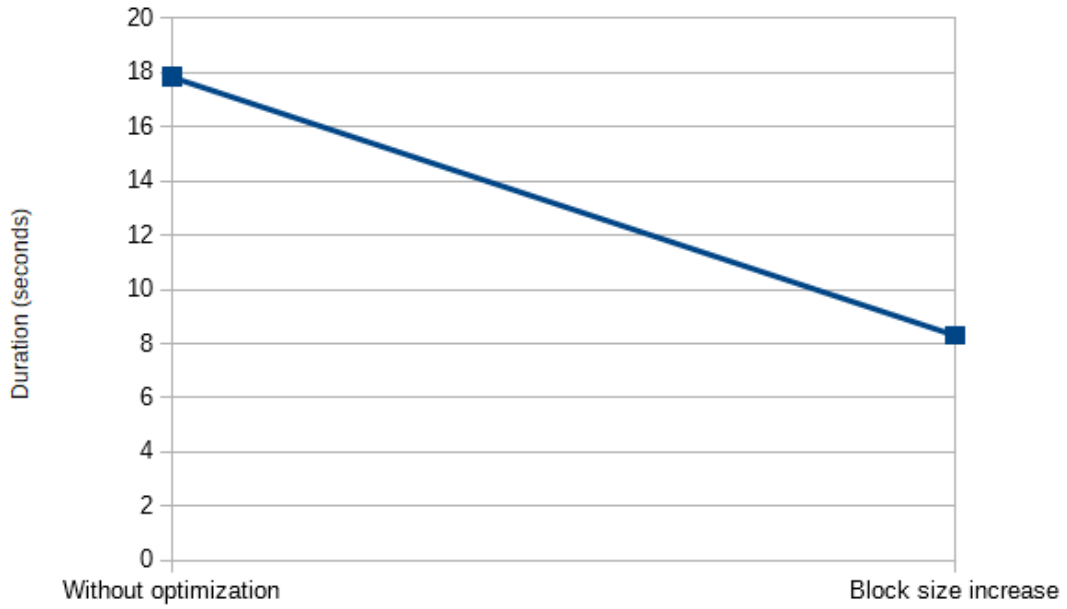


Figure 8.20: NW optimization chart

The total performance gain from the accumulated optimizations of the NW application (based on the non-optimized version) is 2.15.

8.6.5 Analysis

This application was done based on task dependencies. One of the tasks is targeted for the FPGA and the rest are SMP tasks. The trace shown in Figure 8.21 shows how the dependencies between the different tasks are.

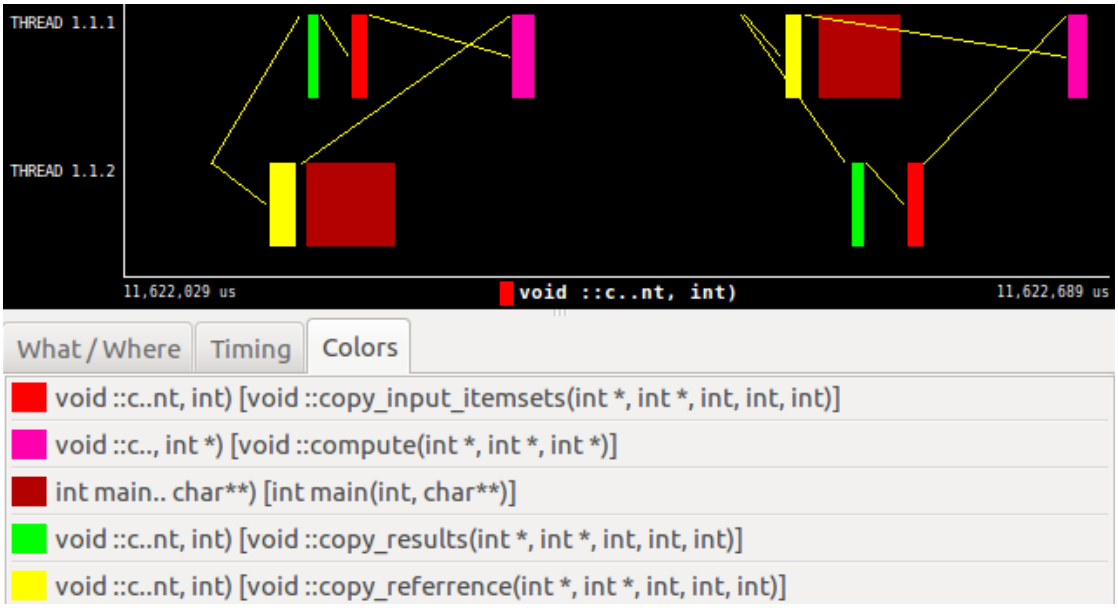


Figure 8.21: NW execution trace without optimization

Once the previous compute task has finished, both copy_results and copy_referrence can start. Once copy_result is finished, copy_input_itemsets can also start. When both copy_referrence and copy_input_itemsets have finished, the FPGA task compute can be executed once again. Note that the compute FPGA tasks take more than it is shown in the trace, specifically they take until the dependence lines to copy_result and copy_referrence functions appear.

The version of the application with bigger block size results in the trace shown in Figure 8.22.

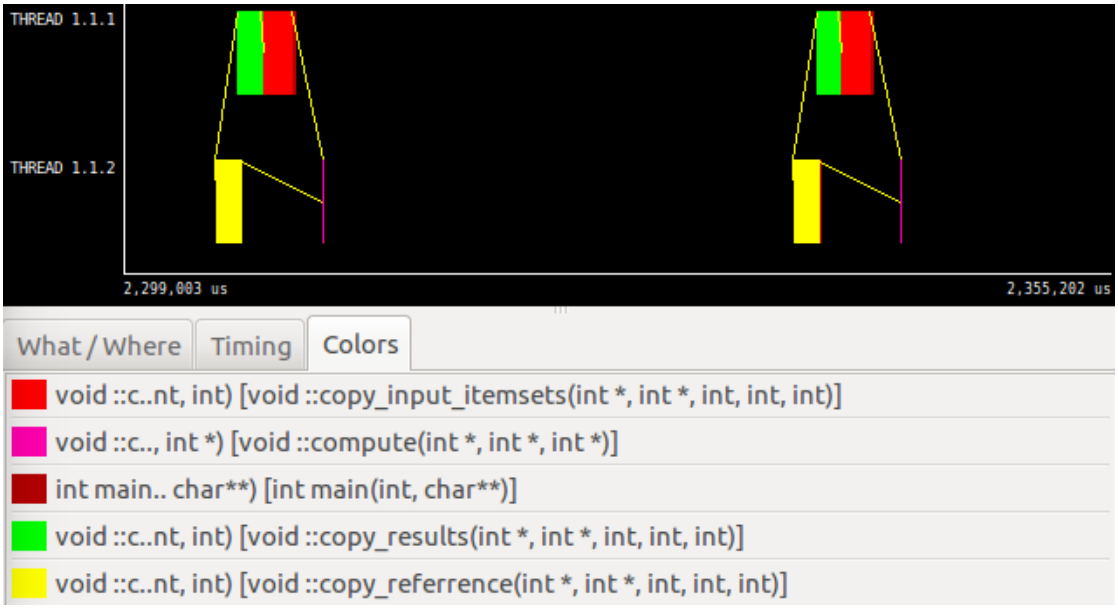


Figure 8.22: NW execution trace with increased block size

The scale of the time window is much larger, but since the new version takes around 512 times the data size of the original (two 2 dimension arrays) it ends up performing better.

8.6.6 Conclusion

While the application can be ported to an FPGA heterogeneous environment, the fact that the data has to be constantly copied from the global array to a local array and vice versa substantially limits its overall performance improvement. All the SMP copy tasks require work which will be dependent on the performance of the host device.

In conclusion, for the application to perform optimally in an FPGA heterogeneous environment, it also needs a good performing host processor.

8.7 Hotspot

8.7.1 Use case analysis

For this application, the input data consisted of two matrices representing the temperature and dissipated power values of each cell respectively. The original

dimensions of the use case fairly small, so the dimensions were increased substantially in order to have more precision when taking optimizations in consideration.

8.7.2 State of the non-optimized application

As explained in the changes made on the original application, it was decided that the FPGA block took the entire number of columns of each row, with a modifiable number of rows to consider for bandwidth tweaking. Since the new use case had the dimensions significantly bigger than the original, it meant that the same amount of rows ended up being a much bigger bandwidth. Because of that, the original number of rows (32) was enough to occupy most of the BRAM from the FPGA.

The duration of the application at this initial stage was of 52.63 seconds.

8.7.3 Loop pipelining

In this case, the two inner loops of the FPGA block were pipelined (shown in Code 8.4).

```
1
2  #pragma HLS ARRAY_PARTITION variable=temp cyclic factor=BLOCK_FACTOR
3  #pragma HLS ARRAY_PARTITION variable=power cyclic factor=BLOCK_FACTOR
4  #pragma HLS ARRAY_PARTITION variable=result cyclic factor=BLOCK_FACTOR
5  for ( chunk = 0; chunk < num_chunk; ++chunk )
6  {
7    int r_start = BLOCK_SIZE_R*(chunk/chunks_in_col);
8    int c_start = BLOCK_SIZE_C*(chunk%chunks_in_row);
9    int r_end =  *rrow_start + r_start + BLOCK_SIZE_R > *max_row ? *max_row : *rrow_start +
     r_start + BLOCK_SIZE_R;
10   int c_end = c_start + BLOCK_SIZE_C > COLS ? COLS : c_start + BLOCK_SIZE_C;
11   if ( (r_start == 0 && *rrow_start == 0) || c_start == 0 || r_end == *max_row || c_end ==
     COLS )
12   {
13     for ( r = r_start; r < r_start + BLOCK_SIZE_R; ++r ) {
14       #pragma HLS PIPELINE II=2
15       for ( c = c_start; c < c_start + BLOCK_SIZE_C; ++c ) {
16         .
17         .
18       }
19     }
20   }
21   else {
22     FLOAT temp_low, temp_high;
23     for ( r = r_start; r < r_start + BLOCK_SIZE_R; ++r ) {
24       #pragma HLS PIPELINE II=2
25       for ( c = c_start; c < c_start + BLOCK_SIZE_C; ++c ) {
26         .
27         .
28       }
29     }
30   }
```

Code 8.4: Pipelined version of the Hotspot FPGA task

Since the number of iterations of both loops were 16, it was decided that the full loop could be pipelined, and so there was no need to add an additional loop

to limit the pipeline factor.

In order to make the read/write operations harder to bottleneck the pipelining process, all three arrays were partitioned in a cyclic way with a factor of 16. This way, every consecutive position in the array pertained in a different memory entity. However, due to both loops also potentially accessing outside of the position based on the iteration, the pipelining process and execution was likely going to encounter some hiccups that could affect performance.

The duration of the application after applying pipelining was of 14.9 seconds, a substantial improvement over the non-pipelined version. The fact that, in the first pipelined loop, the array accesses depended on several 'if' conditionals and most iterations did not enter any of them might have helped reduce the amount of data dependencies and read/write bottlenecks during the execution.

8.7.4 Optimization overview

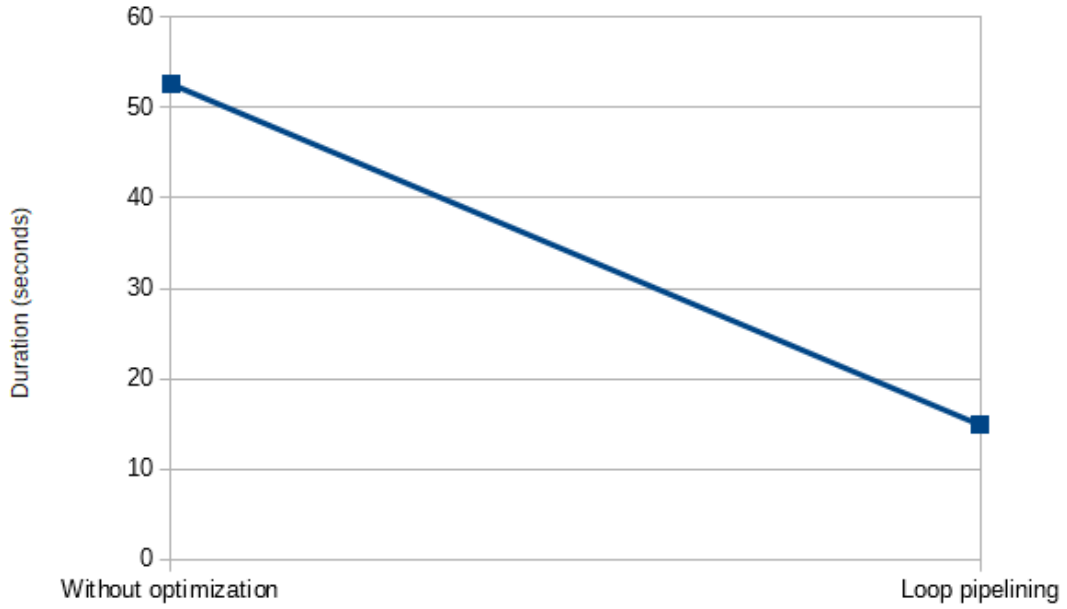


Figure 8.23: Hotspot optimization chart

The total performance gain from the accumulated optimizations of the Hotspot application (based on the non-optimized version) is 3.53.

8.7.5 Analysis

This application was testing with a specified number of iterations of 2. This essentially means that the hotspot algorithm was executed twice. This trace shown in Figure 8.24 represents a single execution of the algorithm.

Figure 8.24: Hotspot execution trace without optimizations

The main function execution in the end of the trace represents the start of the second iteration of the algorithm. We can observe that, while the trace has a large time window compared to the previous applications (half of the application duration), the individual FPGA tasks are still distinguishable. This most likely means that the individual FPGA tasks take a long time to finish.

The pipeline version of the application improves the FPGA block performance by a significant margin (shown in Figure 8.25).

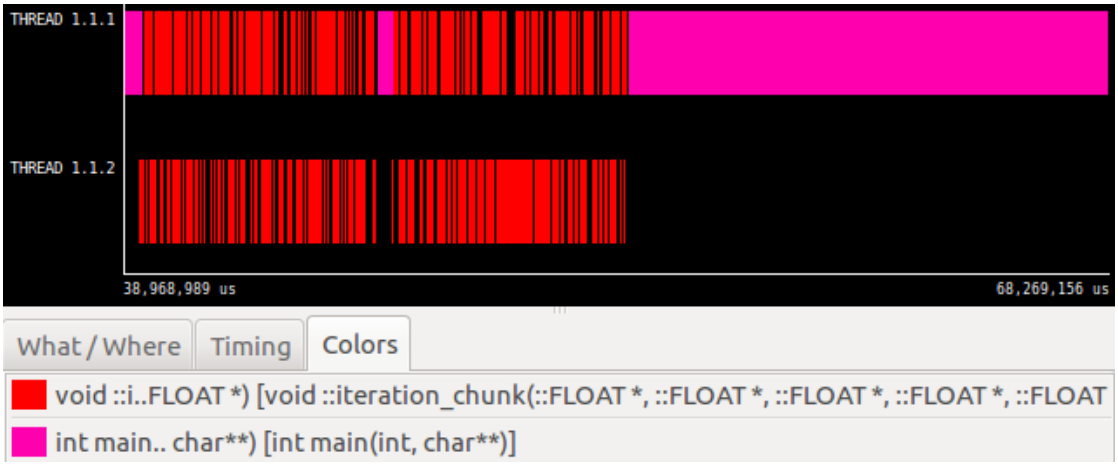


Figure 8.25: Hotspot execution trace with pipelining

With the time that the non-optimized version took to finish a single iteration, the pipelined version finished both and still ended up halfway.

8.7.6 Conclusion

Not only the application can be ported to FPGA heterogeneous computing, but most of its workload can be executed in the FPGA block. Because of that, the application can perform well without the host processor being a hard limiter of its overall performance.

8.8 SRAD

8.8.1 Use case analysis

The input data of this application consisted of an image file in which the algorithm is applied to remove the locally correlated noise. The default use case seemed to produce a long enough execution time for it to be used for optimization purposes.

8.8.2 State of the non-optimized application

The initial block size of the application was 32, since it consisted of several 2 dimension arrays with a fixed second dimension size of 502 (image width). At this stage, the duration of the application was of 35.69 seconds.

8.8.3 Block size increase

While the initial block size was acceptable, it was possible to increase it to a total amount of 128. Such number was decided due to being the last power of 2 to fit in the BRAM of the FPGA device.

At that point, the duration of the application with the specified use case was of 30.55 seconds, not a big improvement but definitely noticeable.

8.8.4 Iteration Pipelining

The initial idea was to pipeline a certain amount of iterations by partitioning the inner loop of each FPGA block in two different loops. However, due to data dependencies between nearby iterations in both dimensions, no reasonable solution came from it. However, since the body of the inner loops were relatively big and could benefit from pipelining, it was decided that we could pipeline single iterations of the loop body instead like shown in Code 8.5.

```
1
2  void work1(fp *image, fp *imgLow, fp *imgHigh, fp *dBuffer, fp *c, fp *q0sqr, int *startRow,
       int *maxRow){
3    .
4    .
5    .
6    #pragma HLS ARRAY_PARTITION variable=image cyclic factor=BLOCK_FACTOR
7    for (j=0; j<BLOCK_SIZE; j++) {
8      for (i=0; i<NR; i++) {
9        #pragma HLS PIPELINE II=1
10         .
11         .
12         .
13     }
14   }
15 }
16
17 void work2(fp *image, fp *c, fp *cHigh, fp *lambda, fp *dBuffer, int *startRow, int *maxRow){
18   .
19   .
20   .
21   #pragma HLS ARRAY_PARTITION variable=c cyclic factor=BLOCK_FACTOR
22   for (j=0; j<BLOCK_SIZE; j++) {
23     for (i=0; i<NR; i++) {
24       #pragma HLS PIPELINE II=1
25       .
26       .
27       .
28     }
29   }
30 }
```

Code 8.5: Pipelined version of the SRAD FPGA task

For the first block, only the 'image' array was partitioned in order to have potential parallel accesses while reading different positions of it. On the other hand, the other block benefited more from a partition of the 'c' array, which was also read several times from different positions. The rest of the arrays on both blocks were only accessed by the specific iteration offset, so there was no need for partitioning them. Both partitions were cyclic, so that consecutive positions were

located in different memory entities.

After applying these changes, the duration of the application was of 7.73 seconds, a substantial improvement over the previous version.

8.8.5 Optimization overview

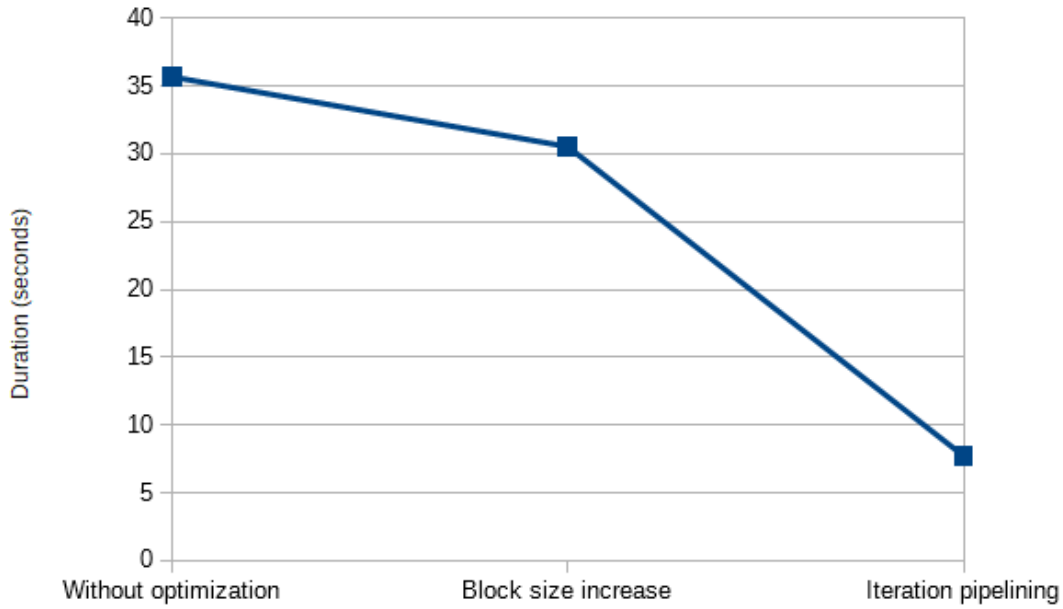


Figure 8.26: SRAD optimization chart

The total performance gain from the accumulated optimizations of the SRAD application (based on the non-optimized version) is 4.62.

8.8.6 Analysis

The trace representing one iteration of the main loop is shown in Figure 8.27.

Figure 8.27: SRAD execution trace without optimizations

There are 2 different group of FPGA tasks separated by a taskwait, as it can be observed in the trace. We can also observe that the work1 task takes twice the amount of time to finish than work2, since it has more computational work. When the block size is increased to 128, the performance does not seem to have improved much (shown in Figure 8.28).

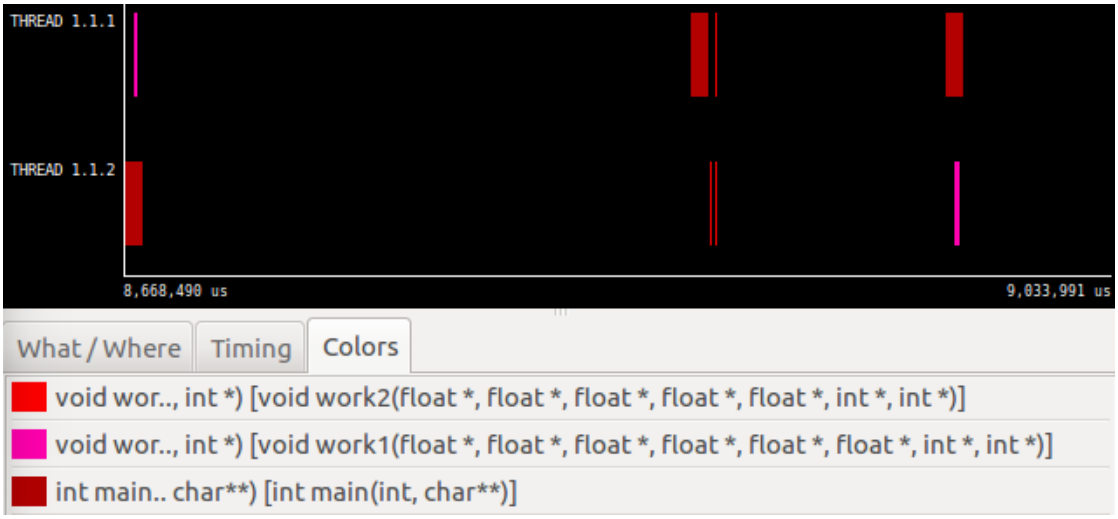


Figure 8.28: SRAD execution trace with increased block size

Less FPGA tasks are created, but the time taken to finish a loop iteration, while a bit better, ends up being very similar. This means that the application

was not having overhead issues with the original bandwidth. However, this could change if the performance of the FPGA blocks themselves were to be improved. Such performance increase is done by pipelining the iteration body of the FPGA blocks, and the trace is shown in Figure 8.29.

Figure 8.29: SRAD execution trace with pipelining

We can observe that the FPGA tasks take much less time to finish than before, thus can fit over 4 times the amount of iterations we had before. While the bandwidth improvement did not improve much at first, it has most likely improved the overall performance after applying the pipelining.

8.8.7 Conclusion

Just like the previous application, most of the workload in the SRAD algorithm is being executed on the 2 FPGA blocks. This means that the application can be executed in an FPGA heterogeneous environment without the host processor hard limiting its overall performance.

8.9 Input/Output bottleneck

By definition, all algorithms have input data to be read in order to perform the computation, along with results to be given after all the work is done. This means that I/O operations can easily be one of the main issues for performance

bottlenecks, especially if the application is dealing with big use cases. Not only the operations are usually sequential, but the time they take to complete is several orders of magnitude bigger than CPU based operations.

In order to reduce these kind of bottlenecks as much as possible, the type of storage needs to be taken in consideration. For example, the usage of high performance Solid State Drives can mitigate the I/O bottleneck in comparison to normal hard drives. RAM disks are also really fast and can be useful if the result can wait to be stored in a persistent way and you need to see it at that instant as fast as possible.

Chapter 9

Sustainability analysis

9.1 Environmental

9.1.1 Project put into production

The environmental impact of undertaking the project is directly proportional to the energy consumed by the personal computer, FPGA device and the server machine attached to it that were used for developing it. Using the average power usage of these three devices and the time each one of them was used for, it is estimated that a total of 180.49 kWh have been consumed during the course of the project.

The creation of an FPGA simulation environment indirectly reduced the environmental impact of the project, since a lot of work which would have been otherwise done using the server machine with the FPGA device was actually doable only using the personal computer. The additional starting environmental cost of the development of the simulator was of 60 hours of the personal computer consumption. On the other hand, while it is hard to accurately approximate it, it would have most likely taken at least the same amount of hours using all three devices at the same time (personal computer to connect to the server, and the server using the FPGA device to test the applications) without the FPGA simulated environment.

9.1.2 Exploitation

This project is not so much about directly improving the use of resources, but rather add to the table an implementation and analysis of another option for the computation of certain algorithms. However, the more options there are, the more accurately people can choose the best candidate depending on their own

preferences. Ultimately, this can result in an overall improvement on the ecological footprint if the FPGA based algorithms chosen have a lower power consumption on average that they would have been on other computation environments.

9.1.3 Risks

Since the objective of this project is to bring to the table another option for algorithm computing, the result can go both ways. There is a possibility that, while being helpful on other aspects, the FPGA based algorithms chosen end up with a higher power consumption, increasing the overall ecological footprint.

9.2 Economic

9.2.1 Project put into production

The quantification of the total costs for the project is stated in the Cost section of the Final mileage chapter. While it was not the original reason for it, the creation of the FPGA simulation environment has definitely lowered the cost of the porting phase of the project, since the usage of the personal computer only instead of all three devices (personal computer, server machine and FPGA device) has lowered the overall power consumption.

I decided to use LaTeX for the final documentation instead of Microsoft Word that I used for the initial documentation, which reduced the cost of using the Microsoft Office license for the amount of hours that it took. If I had to re-do the project with the experience that I now have, I would have made the documentation with LaTeX from the start and save the entirety of the Microsoft Office license. As explained in the Final mileage chapter, the rest of the changes that affected the costs were made because of a change of plans in the middle of the project course.

9.2.2 Exploitation

The cost estimation for the useful life of this project is equivalent to the adaptation cost of the algorithms to the FPGA environment. As previously said, this project adds to the table another option which people might use to save costs over the other options available. Low-end FPGA devices with low capacity can be used for algorithms that don't require a lot of data bandwidth or do not require fast processing, which could cost less than other types of computing (in either or both pricing and power consumption standpoints).

9.2.3 Risks

There is certainly a significant cost of adapting an algorithm computation to an FPGA based environment, since the communication between the FPGA and the host machine tends to be a bit complicated. Moreover, the high-end FPGA devices cost a lot and usually provide less capacity than GPU accelerators. For these reasons, it is possible that some algorithms ported to an FPGA environment end up being more costly than other available options.

9.3 Social

9.3.1 Project put into production

Undertaking the project has definitely helped me prove myself useful by putting a lot of the knowledge I learned during these years into practice. It has also taught me to make proper documentation during the initial phase, in between and in the final phase of a project without leaving it all for the last phase.

I also learned that communication between people involved in a project is critical if you want to progress adequately, and for this reason it is best to do as much as you can between meetings to evaluate the progress made.

9.3.2 Exploitation

The people who will benefit directly from this project are people researching and working in the OmpSs@FPGA department of Barcelona Computing Center, since the main objective was to port the algorithms to that specific computing environment. However, other people who want to use these algorithms for research or commercial purposes can definitely benefit from it as well.

There is a possibility of commercial usage of these algorithms to negatively affect the entity who is using them, since they are not necessarily commercial ready but rather for research purposes. However, this can be easily avoided if proper analysis of the available options is made.

9.3.3 Risks

This project does not create any new social dependencies, but rather provides an implementation and analysis for an additional option for algorithm computing. Because of this, the project does not put people using it in a weak position.

Chapter 10

Final Conclusions and Future Work

While the method used for developing the project changed in a substantial way through its course, the main objective, which was porting some Rodinia Benchmark Suite applications to the OmpSs@FPGA heterogeneous computing environment and optimize some of them to some extent, has been fulfilled.

Even though the FPGA simulation environment was not in the original plan, I think it was the right decision to develop it. Not only it reduced the usage of the server machine and the alpha data device, but it also allowed me to start porting the applications even when the tools for such task were still not given to me. Moreover, it was useful to learn how the runtime worked and the way it sent tasks to the FPGA device.

While FPGA devices are generally more complex and time consuming to develop applications on compared to other alternatives, I think they are an interesting option, especially for low power computing. The ability to synthesize high level language programming into a hardware description language (along with pragmas to help optimize the code for hardware implementation) and automatically generating the bitstream is a big practical improvement over having to use a hardware description language to develop a non-trivial application. It opens a lot of possibilities for implementing applications that would have otherwise been highly impractical to implement for FPGA computing.

On top of that, the OmpSs@FPGA environment allows you to skip the extra steps needed to synthesize an application for FPGA targeting, leaving the developer with only having to implement the FPGA blocks of the application and specify the size of the input/output data on each one.

As future work, further analysis of the benchmark applications ported to the FPGA could be made, including also a power consumption comparison between the host and FPGA versions. The rest of the Rodinia Benchmark Suite applications could also be ported using the experience from this project, as well as other benchmark suites, like the Coral benchmarks [24].

Bibliography

[1] Xilinx. (n.d.)[1]. What is an FPGA? Field Programmable Gate Array. Retrieved February 25, 2019, from https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html

[2] Xilinx. (2011). Power Methodology Guide, 786 (pp. 11–12). Retrieved from https://www.xilinx.com/support/documentation/sw_manuals/xilinx13_1/ug786_PowerMethodology.pdf

[3] Xilinx. (2017). High-Level Synthesis Flow on Zynq using Vivado. Retrieved February 25, 2019, from https://www.xilinx.com/support/university/vivado/vivado-workshops/Vivado-high-level-synthesis-flow-zynq.html

[4] Barcelona Supercomputing Center. (n.d.). The OmpSs Programming Model — Programming Models @ BSC. Retrieved February 25, 2019, from https://pm.bsc.es/ompss

[5] Filgueras, A., Gil, E., Alvarez, C., Jimenez, D., Martorell, X., Langer, J., & Noguera, J. (2013). Heterogeneous tasking on SMP/FPGA SoCs: The case of OmpSs and the Zynq. In 2013 IFIP/IEEE 21st International Conference on Very Large Scale Integration (VLSI-SoC) (pp. 290–291). IEEE. https://doi.org/10.1109/VLSI-SoC.2013.6673293

[6] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., & Skadron, K. (2009). Rodinia: A benchmark suite for heterogeneous computing. In 2009 IEEE International Symposium on Workload Characterization (IISWC) (pp. 44–54). IEEE. https://doi.org/10.1109/IISWC.2009.5306797

[7] Barcelona Supercomputing Center. (n.d.). OmpSs@FPGA — Programming Models @ BSC. Retrieved February 25, 2019, from https://pm.bsc.es/ompss-at-fpga

[1] No publish date specified.

[8] (1) PageGroup (2018). Tendencias del mercado laboral - Tecnología. Retrieved March 11, 2019, from https://www.pagepersonnel.es/sites/pagepersonnel.es/files/PG_ER_IT_2018.pdf (2) Page Personnel (2016). Estudios de remuneración 2016 - Comercial. Retrieved March 11, 2019, from http://www.pagepersonnel.es/sites/pagepersonnel.es/files/ER_comercial16.pdf

[9] Ministerio de Economía y Hacienda. (2004). BOE.es - Documento consolidado BOE-A-2004-14600. Retrieved March 11, 2019, from https://www.boe.es/eli/es/rd/2004/07/30/1777/con

[10] (1) AVNET. (2012). ZynqTM Evaluation and Development Hardware User's Guide. Retrieved March 11, 2019, from https://reference.digilentinc.com/_media/zedboard:zedboard_ug.pdf (2) AXIOM. (n.d.). AXIOM Project – Agile, eXtensible, fast I/O Module for the cyber-physical era. Retrieved March 11, 2019, from http://www.axiom-project.eu/

[11] Endesa S.A. (2019). La tarifa PVPC — Facturació per hores — ENDESA CLIENTS. Retrieved March 11, 2019, from https://www.endesaclientes.com/pvpc-preu-voluntari-petit-consumidor-mercat-regulat

[12] Xilinx. (2015). Xilinx 7 Series FPGA Power Benchmark Design Summary Application-centric Benchmarking Process, (May). Retrieved September 16, 2019, from https://www.xilinx.com/publications/technology/power-advantage/7-series-power-benchmark-summary.pdf

[13] ESIOS. (2019). PVPC — ESIOS electricidad · datos · transparencia. Retrieved September 20, 2019, from https://www.esios.ree.es/es/pvpc

[14] University of Virginia. (2018). k-nearest_neighbors [Rodinia]. Retrieved July 30, 2019, from https://rodinia.cs.virginia.edu/doku.php?id=k-nearest_neighbors

[15] University of Virginia. (2018). pathfinder [Rodinia]. Retrieved July 30, 2019, from https://rodinia.cs.virginia.edu/doku.php?id=pathfinder

[16] University of Virginia. (2018). needleman-wunsch [Rodinia]. Retrieved July 30, 2019, from https://rodinia.cs.virginia.edu/doku.php?id=needleman-wunsch

[17] University of Virginia. (2018). hotspot [Rodinia]. Retrieved July 30, 2019, from https://rodinia.cs.virginia.edu/doku.php?id=hotspot

[18] University of Virginia. (2018). srad [Rodinia]. Retrieved July 30, 2019, from https://rodinia.cs.virginia.edu/doku.php?id=srad

[19] University of Virginia. (2018). myocyte [Rodinia]. Retrieved July 30, 2019, from https://rodinia.cs.virginia.edu/doku.php?id=myocyte

[20] Szafaryn, L. G., Skadron, K., & Saucerman, J. J. (2009). Experiences Accelerating MATLAB Systems Biology Applications. Biomedicine in Computing: Systems, Architectures, and Circuits (BiC), 1–4.

[21] Xilinx Inc. (2018). pragma HLS array_partition. Retrieved September 16, 2019, from https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/gle1504034361378.html

[22] Xilinx Inc. (n.d.). Increasing Local Memory Bandwidth. Retrieved September 16, 2019, from https://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_2/sdsoc_doc/topics/calling-coding-guidelines/concept_increasing_local_memory_bandwidth.html

[23] Xilinx Inc. (2018). pragma HLS pipeline. Retrieved September 16, 2019, from https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/fde1504034360078.html

[24] Oak Ridge, Argonne, Livermore, "Coral Collaboration", https://asc.llnl.gov/CORAL-benchmarks/